Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems

Xavier Palomo Teruel

UNIVERSITAT ROVIRA i VIRGILI

Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems

Xavier Palomo Teruel

DOCTORAL THESIS

2025

Programa de Doctorat en Enginyeria Informàtica i Matemàtiques de la Seguretat

UNIVERSITAT ROVIRA i VIRGILI

Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems

DOCTORAL THESIS

Xavier Palomo Teruel

Supervisor
Dr. Carlos Molina Clemente
Departament d'Enginyeria Informàtica i Matemàtiques

Tarragona, Spain, 2025

UNIVERSITAT ROVIRA I VIRGILI
Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems
Xavier Palomo Teruel

UNIVERSITAT ROVIRA i VIRGILI

(Català) FAIG CONSTAR que aquest treball, titulat "**Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems**", que presenta **Xavier Palomo Teruel** per a l'obtenció del títol de doctor o doctora, s'ha realitzat sota la meva direcció al **Departament d'Enginyeria Informàtica i Matemàtiques** d'aquesta universitat.

(Espanyol) HAGO CONSTAR que el presente trabajo, titulado "**Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems**", que presenta **Xavier Palomo Teruel** para la obtención del título de doctor o doctora, se ha realizado bajo mi dirección en el **Departament d'Enginyeria Informàtica i Matemàtiques** de esta universidad.

(Anglès) I STATE that the present work, entitled "**Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems**" and presented by **Xavier Palomo Teruel** for the degree of doctor, has been carried out under my supervision at the **Departament d'Enginyeria Informàtica i Matemàtiques** of this university.

Tarragona, 17 de junio de 2025

La direcció de la tesi doctoral
La dirección de la tesis doctoral
The Doctoral Thesis Supervisor/s

MOLINA CLEMENTE CARLOS MARIA - 52460503X

Firmado digitalmente por MOLINA CLEMENTE CARLOS MARIA - 52460503X
Fecha: 2025.06.17 11:37:50 +02'00'

[signatura] / [firma] / [signature] [signatura]
Dr. Carlos Molina Clemente

Resum

Aquesta tesi doctoral se centra en un dels reptes més rellevants dels sistemes temps real sobre multiprocessadors: obtenir una estimació fiable i segura del *Worst-Case Execution Time (WCET)* quan diverses tasques comparteixen recursos. L'ús creixent de processadors multinucli en àmbits de seguretat crítica —com l'aeronàutica, l'automoció o l'espai— exigeix tècniques d'anàlisi que tinguin en compte la interferència entre nuclis i garanteixin la predictibilitat temporal.
Es presenten dues metodologies innovadores complementàries:

1. La primera formula el problema mitjançant Programació Lineal Entera (ILP) i modela, de manera exhaustiva, totes les interaccions possibles als recursos compartits. El resultat és un WCET globalment segur, adequat quan cal la màxima certesa a nivell de sistema *(makespan)*.

2. La segona segueix un procés iteratiu que ofereix garanties a nivell de tasca. En centrar-se en cada fil d'execució de manera individual, introdueix més pessimisme sobre el *makespan* global, però simplifica l'anàlisi i facilita l'aplicació pràctica en plataformes industrials on la granularitat per tasca és prioritària.

Per alimentar amb dades fiables ambdós enfocaments, s'han dissenyat microbenchmarks en codi assemblador capaços de provocar, de forma controlada i densa, esdeveniments com *missos* de memòria cache o congestió del bus. Aquests microbenchmarks permeten validar la precisió del procés de *profiling* tant en plaques amb unitats de Performance Monitoring Unit (PMU) com en aquelles que no en disposen. En aquest segon escenari —habitual en maquinari aeroespacial o de baix cost— la tesi proposa i implementa una solució software que, aprofitant registres de depuració i una anàlisi detallada de l'arquitectura, infereix de manera conservadora els comptadors d'esdeveniments i possibilita igualment el càlcul de WCET i l'anàlisi temporal.

Les contribucions d'aquesta tesi —les dues metodologies de WCET, el conjunt de microbenchmarks i els procediments de perfilatge amb i sense PMU— ofereixen un marc sòlid per millorar la predictibilitat i la seguretat dels sistemes temps real multinucli, tot reduint la bretxa entre les anàlisis teòriques i la seva aplicació en plataformes amb fortes restriccions industrials.

Acknowledgements

Completing this Thesis has been a challenging and rewarding journey, and I am deeply grateful to the many people who have supported me along the way.

First and foremost, I want to express my gratitude to my parents, Elvira and Ángel. Your unwavering belief in the value of education has been the cornerstone of my academic path. You always strived to give me every opportunity, encouraged me to pursue what I enjoyed, and supported me without hesitation. This work would not have been possible without your constant support, and for that, I am forever grateful.

I am deeply thankful to my supervisors and the colleagues who contributed directly to my research and the papers that form the foundation of this Thesis. Your feedback, guidance, and collaboration have been invaluable throughout the process. Our discussions and your insights pushed me to think deeper and aim higher, and I owe much of this work to your contributions.

To my colleagues at Barcelona Supercomputing Center, thank you for creating a supportive environment where I could thrive. From engaging technical discussions to sharing experiences and frustrations, you have made this journey not only manageable but also enjoyable. The friendships built during this time will remain with me long after this Thesis.

I also want to acknowledge my friends and family beyond academia, who have kept me grounded. To Julia, thank you for your patience and understanding, and for supporting me quietly in your own way. It has meant more to me than I could express.

Finally, I am grateful to the broader academic community that has inspired me and contributed to this work through their own research and ideas. I hope that this Thesis, in turn, adds some value to the field and the efforts of those working in it.

This Thesis is the result of hard work, collaboration, and the incredible support of everyone mentioned above. I am truly grateful to all of you for being part of this journey. Thank you for making it possible.

Xavier Palomo Teruel
Tarragona, Spain
2025

Abstract

This Thesis addresses one of the critical challenges in real-time multicore systems: the accurate and safe determination of Worst-Case Execution Time (WCET). With the increasing adoption of multicore processors in safety-critical domains, ensuring the timing predictability of tasks under contention becomes paramount. To tackle this, this work proposes two novel methodologies for WCET analysis tailored to multicore architectures.

The first is an Integer Linear Programming (ILP) formulation that computes WCET by rigorously modeling all possible contention scenarios. This method ensures system-wide guarantees by accounting for the worst-case interleaving of tasks on shared resources. The second is an iterative approach that provides task-level timing guarantees, trading off some system-level pessimism for more practical and less conservative bounds. Both approaches are designed to be flexible and applicable across a wide range of multicore platforms.

To support the proposed methods, this Thesis also develops and employs microbenchmarks specifically crafted to stress different parts of the hardware and generate a dense occurrence of targeted events, such as cache misses or memory accesses. These benchmarks are implemented at the assembly level to maximize event precision and efficiency, allowing us to rigorously validate and verify the accuracy of the profiling process.

In addition, we detail the methodology for deriving performance events using performance monitoring counters (PMCs) and the performance monitoring unit (PMU). For platforms lacking a PMU, a robust mathematical approach is introduced to conservatively estimate the necessary events. Furthermore, we demonstrate the development of a software solution capable of accurately deriving these events on a board without a PMU, leveraging debug features and a deep architectural analysis to infer the required performance metrics.

Through the integration of these techniques, this Thesis not only proposes innovative methods for timing analysis but also ensures their applicability across constrained industrial platforms. The contributions made here aim to enhance the predictability and safety of multicore systems in real-time environments, bridging the gap between theoretical timing analysis and practical implementation challenges.

UNIVERSITAT ROVIRA I VIRGILI
Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems
Xavier Palomo Teruel

Contents

List of Figures

List of Tables

UNIVERSITAT ROVIRA I VIRGILI
Advancing Multicore Timing Analysis: Modeling Contention and Deriving WCET Bounds in Cyclic Executive Systems
Xavier Palomo Teruel

Chapter 1

Introduction

In the realm of (critical) real-time embedded systems, ensuring that systems meet stringent timing requirements is not just a matter of performance—it is a fundamental aspect of safety and reliability. Whether in aerospace, automotive, or industrial applications, the correct operation of these systems hinges on their ability to perform within tightly bounded execution times. The increasing complexity of modern embedded systems, coupled with the shift towards multicore processors, has introduced new challenges in achieving and verifying these timing guarantees.

This thesis addresses these challenges by exploring advanced methods for profiling and timing analysis, specifically within the context of multicore CRTES. To that extend, this Thesis proposes two novel timing analysis methods for these systems: an iterative method and an Integer Linear Programming (ILP) based analysis, both of which are designed to derive worst-case delay bounds in systems where performance monitoring units (PMUs) are available. Additionally, a software solution is proposed for a widely used space-grade multicore platform, the GR712RC, which does not count on a PMU.

The core objective of this Thesis is to demonstrate how robust timing analysis can be conducted even in environments with limited hardware support and the proposal of different approaches to calculate the Worst-Case Execution Time (WCET). By leveraging a deep understanding of the processor architecture and innovative software-based solutions, this work aims to expand the applicability of advanced timing analysis techniques to a broader range of industrial scenarios.

1.1 Critical Real-Time Embedded Systems

Critical Real-Time Embedded Systems (CRTES) represent a specialized category of systems that are integral to the functioning of crucial applications, particularly in domains where failure or timing inaccuracies can lead to catastrophic outcomes. The defining characteristic of CRTES is their stringent requirement for timing correctness, which is as critical as the functional correctness of the system.

At the core of CRTES is the principle that these systems must adhere to strict timing

constraints. This encompasses not only the necessity for responses to be produced within a specific time frame but also that these responses must be predictably correct in both their temporal and functional aspects. Timing correctness in CRTES implies that the system must respond to inputs or events within a predefined time interval, known as the deadline. The failure to meet these deadlines can have severe, sometimes irreversible, implications in real-world scenarios, such as in avionics, automotive safety systems, medical devices, and industrial control systems.

In CRTES, the embedded nature signifies that the system is a critical part of a larger mechanical or electrical system. This integration often requires that the embedded system is highly optimized for size, power, and efficiency, and operates under constraints of limited computing resources. Despite these limitations, the system must guarantee real-time responsiveness and reliability.

The criticality of these systems is underscored by their deployment in environments where safety, security, and reliability are paramount. The design and development of CRTES, therefore, involve rigorous methodologies to ensure that both the hardware and software components can reliably meet the temporal specifications under all operating conditions. This includes worst-case execution time analysis, rigorous testing, and often adherence to specific industry standards and certifications.

1.2 Challenges in CRTES Timing Analysis

The evolution of CRTES towards multicore architectures marks a pivotal shift in response to the diminishing scalability of single-core processors, as predicted by Moore's Law. This transition, while offering significant enhancements in computational capacity, introduces a new spectrum of complexities for timing analysis that were not prevalent in single-core environments.

1. **Enhanced Computational Capacity with New Challenges**: Multicore processors, by virtue of parallel processing capabilities, promise considerable improvements in processing power and efficiency. However, this advantage comes at the cost of increased complexity in system behavior. In CRTES, where timing precision is paramount, this complexity manifests as a substantial challenge for timing analysis.

2. **Inter-Core Interference and Shared Resources**: One of the fundamental issues introduced by multicore architectures is the contention for shared resources. Cores in a multicore system often share crucial resources such as caches, memory buses, and I/O interfaces. When multiple cores attempt to access these shared resources simultaneously, contention occurs, leading to delays and unpredictable system performance. This phenomenon significantly complicates the timing analysis, as it introduces a layer of uncertainty in execution times.

3. **Complexity in Timing Predictability**: In CRTES, ensuring that all tasks meet their strict deadlines is critical. The contention in shared resources in multicore

systems makes it challenging to accurately predict the execution time of each task. Unlike single-core systems where task execution times are relatively predictable, multicore systems exhibit a wide range of execution variances due to inter-core interference, making the timing analysis more intricate.

4. **Deterministic Behavior vs. Performance Gains**: The move to multicore systems in CRTES poses a trade-off between the deterministic behavior required for real-time applications and the performance gains offered by parallel processing. Achieving a balance where system predictability is maintained without significantly undermining the performance benefits of multicore processing is a key challenge in CRTES design and development.

5. **Architectural Considerations for Real-Time Systems**: The architectural design of multicore systems plays a crucial role in their suitability for CRTES. Factors such as cache architecture, bus arbitration mechanisms, and core-to-core communication protocols need careful consideration. The design must address the potential for resource contention while maximizing the system's real-time responsiveness.

1.2.1 Contention of Shared Resources

Multicore systems inherently involve multiple processing units sharing common resources, which include memory buses, caches, and I/O interfaces. This architectural design, while beneficial for parallel processing and enhanced computational efficiency, introduces the challenge of resource contention. Contention occurs when simultaneous demands from multiple cores converge on these shared resources, leading to conflicts and consequential waiting times. This phenomenon is particularly critical in the context of CRTES, where it directly impacts both the predictability and the overall performance of the system.

In the domain of CRTES, especially those operating under hard real-time constraints, the predictability of system behavior is not just a requirement but a necessity. These systems are often deployed in environments where failure to adhere to strict deadlines could result in catastrophic outcomes. For instance, in avionics or medical systems, any overrun of the execution window, however slight, may lead to irrevocable system failures or hazardous situations. Hence, the ability to accurately characterize and predict the extent and impact of contention in shared resources becomes a cornerstone in the design and evaluation of CRTES.

The complexity of this challenge is multifold. Firstly, the diverse nature of shared resources and their varying contention scenarios complicate the modeling and analysis process. Each type of shared resource, from memory buses to caches, has unique characteristics and contention behaviors. Furthermore, the dynamic nature of real-time systems, where task priorities and execution schedules may vary, adds an additional layer of complexity to the contention analysis.

Secondly, the requirement for high accuracy in the prediction of contention delays is paramount. In hard real-time systems, the tolerance for timing inaccuracies is minimal.

Effective contention analysis must, therefore, not only identify potential contention scenarios but also quantify the expected delays with a high degree of precision. This precision is crucial for ensuring that all tasks within the system can be reliably scheduled to meet their deadlines without any risk of overrun.

Finally, developing methods to mitigate or manage contention in shared resources is an essential aspect of research in this area. Techniques to reduce contention, such as intelligent task scheduling or hardware modifications like cache partitioning, are actively explored to enhance the predictability and reliability of CRTES. However, these methods must be balanced against other system requirements, such as computational efficiency and resource utilization, to ensure the overall efficacy of the system.

1.2.2 Lack of Performance Monitoring Capabilities

In the realm of CRTES, conducting an effective timing analysis hinges significantly on the availability of advanced hardware monitoring capabilities. Essential tools for this purpose are the Performance Monitoring Unit (PMU) and the associated Performance Monitoring Counters (PMC), which serve as the backbone for analyzing and understanding the complex behaviors of multicore systems. These tools are instrumental in providing detailed data on various system performance metrics, such as cache misses, branch mispredictions, and execution stalls—each playing a critical role in the precise calculation of system timing.

The challenge, however, emerges when such sophisticated monitoring capabilities are not universally present in industrial-grade embedded platforms. A significant portion of embedded systems, especially those deployed in industrial settings, lack these comprehensive performance monitoring facilities. This deficit is not merely a technical inconvenience but a substantial impediment to achieving accurate and granular insights into system performance, which are indispensable for meticulous timing analysis.

The absence of PMUs and PMCs in many CRTES leads to a critical gap in the ability to obtain low-level performance data. This gap hinders the capability of system engineers and developers to precisely characterize and predict system behavior under various operating conditions. The repercussions of this limitation are particularly pronounced in scenarios where understanding the intricate details of system performance is crucial for ensuring compliance with the stringent timing requirements of real-time operations.

In response to this challenge, developers are often compelled to resort to alternative methods for system performance analysis. These alternatives, however, frequently fall short of providing the level of detail and accuracy offered by PMUs and PMCs. Techniques such as software-based monitoring or model-based analysis, while useful, often involve trade-offs in terms of granularity, accuracy, or overhead. This situation underscores a critical need in the CRTES field for more universally accessible and sophisticated hardware monitoring tools or for the development of innovative methodologies that can compensate for the lack of such tools.

The absence of robust performance monitoring capabilities in many industrial-grade CRTES thus poses a significant challenge in the field, necessitating creative and effective solutions to bridge the gap between the need for precise timing analysis and the limitations

of existing system monitoring facilities.

1.2.3 Predictability in Multicore Environments

Ensuring predictability in multicore CRTES environments encompasses a series of intricate challenges. This complexity extends beyond understanding resource contention to include managing execution path variations, interference patterns, and the multifaceted interactions between hardware and software components. The inherent non-determinism in these factors complicates the assurance of consistent timing compliance under diverse operational conditions.

1. **Variability in Execution Paths**: The execution path in multicore systems can exhibit significant variability due to cache behavior, branch prediction, and varying system loads. These fluctuations impact the predictability of execution times, necessitating advanced analytical methods to accommodate a broad range of operational scenarios.

2. **Interference Patterns Analysis**: Interference patterns, where tasks on different cores interact through shared resources, play a crucial role in timing analysis. Characterizing these patterns accurately is vital yet challenging due to their complexity and variability.

3. **Hardware and Software Interactions**: The interaction between hardware features and software in multicore systems adds layers of complexity to predictability. Out-of-order execution, speculative execution, and memory hierarchy design can all impact software performance in unpredictable ways, demanding comprehensive modeling for accurate prediction.

4. **Non-Deterministic Nature of Multicore Systems**: Multicore systems inherently exhibit non-determinism, a result of concurrent task executions and shared resource management. This randomness in task execution and resource utilization stands as a primary challenge to predictability in multicore CRTES.

Addressing these challenges calls for the deployment of sophisticated modeling and simulation tools, capable of accurately reflecting the behavior of multicore systems under a spectrum of conditions. Moreover, the development of innovative scheduling algorithms and architectural optimizations can aid in mitigating the unpredictability inherent in these systems, steering them towards reliable and consistent performance within their timing constraints.

1.3 Selecting the Profiling Solution for Timing Analysis

For timing analysis, profiling solutions typically rely on extracting pertinent information while the program under analysis is running. This relevant information encompasses all hardware events that influence timing behavior. Modern COTS hardware platforms are often equipped with a PMU. The PMU allows access to various hardware event counters through PMCs, which can be configured to monitor specific hardware events. These hardware events generally either count occurrences (e.g., cache misses) or measure the impact in terms of cycles (e.g., stall cycles). It's important to highlight that monitoring support was originally developed for low-level hardware debugging or high-level performance optimization, which are tasks with fundamentally different goals from timing analysis.

Despite the availability of multiple profiling solutions, no standardized solution exists for multicore timing analysis due to the diverse requirements for timing analysis. Industrial users must select the profiling solution that best aligns with their specific qualification and certification needs. Below, we discuss the key considerations that should inform the choice of a profiling method.

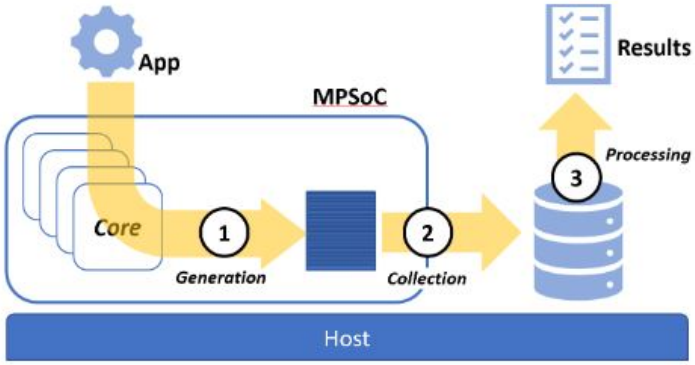


Figure 1.1: Measurement process breakdown

1.3.1 Steps in the Measurement Process

The fundamental unit of timing measurements is known as an observation item (oItem), which typically includes the values of one or more event counters at a specific moment. Drawing from our experience, we present in Figure 1 a high-level overview of a reference measurement framework, which consists of three main steps:

1. **Generation of observation items**. This step involves capturing a snapshot of selected event counters (usually including a timestamp and a set of events) and storing it in a designated memory location. The method for triggering the generation of

oItems, commonly implemented by marking specific points in the program (instrumentation points), is a key differentiator. We classify these methods into software and hardware instrumentation. Additionally, the storage location for oItems can vary, ranging from shared memory areas to dedicated on-chip devices. The choice of oItem generation method significantly impacts the measurement framework in terms of intrusiveness and required hardware support [60, 11].

2. **Collection of observation items**. The subsequent step involves gathering the stream of generated oItems to facilitate further processing and analysis. Exporting these observations from the on-chip storage is a delicate process in terms of intrusiveness. This can be achieved through either in-band or out-of-band solutions [59]. In-band solutions use the same hardware interconnect as the application and standard debugger support to collect oItems. In contrast, out-of-band solutions utilize dedicated trace collection mechanisms. A critical consideration in this step is that the volume of oItems can become substantial depending on the analysis scope. In such cases, in-band solutions may become overly resource-intensive and impractical, while out-of-band approaches depend on the availability of the necessary hardware support [60, 67].

3. **Processing of observation items**. This step distinguishes between on-line and off-line processing approaches. On-line approaches [28] process the profiling information immediately after it is collected from the target hardware, whereas off-line approaches handle the data in a single block after collection. The primary difference between these approaches lies in the scope and size of the analysis: off-line approaches require storing larger amounts of data but minimize interference during oItem collection, as it only occurs at the end of the experiment. On-line approaches reduce the data storage requirements but may increase interference if a dedicated out-of-band tracing solution is unavailable.

1.3.2 Characterization of a Profiling Approach

In this subsection, we identify and discuss key aspects to consider when selecting a profiling solution.

Intrusiveness of Instrumentation. From an analysis standpoint, it is crucial that observations are collected without significantly distorting the system's behavior. This potential distortion is known as the probe effect, a common pitfall in experimental processes. Ideally, less intrusive approaches are preferred, but the actual probing configuration often depends on the hardware and tool support available for a specific target. Intrusiveness is mainly influenced by the instrumentation approach, which can be software-based or hardware-based. Software instrumentation [68] is advantageous due to its ease of integration with any target and development tool-chain, applicable to various analysis scenarios with minimal effort. However, the instrumentation code must be extremely lightweight to avoid significantly impacting system behavior [31]. Software frameworks provide a straightforward method for accessing memory-mapped hardware monitoring counters with minimal

overhead, ensuring reuse across different configurations [39]. While standardized profiling APIs are available for mainstream and high-performance targets [55, 19], no mature standard exists for embedded systems. On the other hand, hardware instrumentation requires specific debug support [60, 11] through advanced Debug Support Units (DSUs), enabling zero-intrusiveness oItem collection [28]. DSUs are increasingly common in embedded platforms, but despite some standardization efforts [60, 11], the instrumentation framework often needs to be tailored to the specific target and debug device.

Intrusiveness of Data Collection. The method used for storing collected oItems and the data collection mechanism itself also influence the intrusiveness of the profiling approach. In-band profiling, typically used by software instrumentation frameworks and relying on local memory for storing oItems, offers straightforward integration but can introduce significant interference, especially with fine-grained instrumentation. Out-of-band solutions are necessary when performance counters are not memory-mapped or are inaccessible to the user. These solutions are typically employed by advanced debugging devices and tools to provide non-intrusive profiling. Out-of-band solutions also allow for the collection of larger data volumes without risking system interference. However, these hardware solutions are often specific to a target family and thus offer limited reuse.

Scope of the Timing Analysis. The scope of the analysis significantly impacts the effectiveness of a profiling solution. Various dimensions need to be considered: instrumentation granularity, type of collected information, and the execution model. Conventional measurement-based analysis typically relies on end-to-end measurements at the task or partition level. Hybrid analysis approaches [53, 68] may consider finer-grained analyses, correlating measurements over small code blocks with the program's structural information. The analysis scope affects the frequency of oItem generation, which can increase the profiling mechanism's intrusiveness. For multicores, the solution may need to profile multiple cores simultaneously, exacerbating intrusiveness and complicating oItem generation and collection. Alternatively, one could rely on oItems extracted from isolated execution and analytically define conservative assumptions (e.g., on overlapping contender requests) to enforce worst-case interference scenarios [24, 44, 25]. Regarding the type and amount of information collected at each instrumentation point, traditional timing analysis approaches typically associate a unique program point with a timestamp. Advanced multicore system approaches [59] may require extensive performance monitoring unit data, affecting both intrusiveness and the measurement process since not all events can be tracked simultaneously due to limited performance counters. Profiling usually involves single event counts, where oItems store information on the number of events (or cycles related to an event) during the observation period. In multicore systems, some events are interdependent, and simple event counts may not suffice. For example, bus access latency can be influenced by recent events. In such cases, full traces are needed, significantly increasing the amount of data to be collected and processed. Large data volumes challenge in-band solutions and may exceed the processing capabilities of on-line processing approaches [28]. In-band methods that cannot ensure low intrusiveness while collecting events and timing information can compromise the latter's accuracy by including the profiling cost.

Hardware Support. The availability of runtime monitoring hardware support is

crucial in selecting a profiling approach. Since debug support is not typically a primary driver for platform selection, choosing a profiling solution is sometimes limited by available support. Solutions range from fully integrated on-chip options to more complex off-chip solutions using external devices. Fully integrated on-chip solutions represent the baseline for hardware profiling, typically involving a specialized debugger developed by the hardware manufacturer, executed on the host platform and connected to the target via a generic communication port (e.g., JTAG, GPIO, Ethernet). This approach is cost-effective and minimally demanding for profiling, supporting hardware instrumentation but limited to in-band tracing, where oItems are stored in on-chip memory and moved to host memory for off-line processing. While on-line processing is technically possible, it is discouraged due to the limited bandwidth of in-band solutions. Specialized external hardware can also be used for oItem extraction [67]. On the other end of the spectrum, specialized external hardware provides advanced tracing capabilities through an external debugging device connected to the target via a generic or target-specific probe and high-bandwidth protocols (e.g., AURORA). The host machine configures the profiling process and selects oItems. Although relatively expensive, specialized hardware debug support enables out-of-band tracing. Commercial debugging devices (e.g., Lauterbach) typically allow some reuse across target families by changing the terminal probe. Advanced out-of-band and debug hardware support offer the best zero-intrusiveness profiling solution [28]. Additionally, specific hardware support affects the resolution of timing measurements, as available implementations may support different clock frequencies for timestamps, not always matching the cycle clock frequency. Throughout our experiments, the GR712RC board operated at its default clock frequency of 80MHz.

Safety-Related Requirements. Profiling solutions must also address safety certification considerations. For software instrumentation, the instrumentation code (usually a lightweight macro) must be carefully assessed from a certification perspective. The instrumentation code may need to remain in the final software configuration since analysis results are valid for the instrumented program, but some certification standards may prohibit deactivated or non-functional code in the target program. The approach depends on the specific industrial domain and certification standard [31].

Other Industrial Requirements. Practical concerns can also influence the selection of a hardware profiling approach. These include technical aspects such as integration with specific host OS or development tools and processes. The primary industrial criterion is the cost/benefit ratio: the profiling framework is evaluated based on the solution's cost and the induced costs (training, usage, etc.). The optimal solution depends on the profiling requirements and available target platform support. Additionally, in the long term, the portability of the profiling solution is a major concern: having to rethink and redeploy established profiling tools and practices due to a change in processor family or RTOS is impractical.

1.4 Objective of this Thesis

The primary objective of this thesis is to advance the field of CRTES in multicore environments by providing innovative and more effective solutions, transcending the limitations of existing methodologies. The thesis is structured to address the complex challenges associated with CRTES, focusing on improving predictability and reliability under specific system models and assumptions related to scheduling and system architecture, which will be elaborated in subsequent sections.

Central to this research is the development of robust solutions that ensure reliable performance guarantees at both task and system levels. For the task level, the thesis introduces an iterative approach designed to optimize execution time analysis in scenarios where resource contention significantly influences system performance. This approach is tailored to meet the unique demands of applications that require stringent task-level guarantees.

At the system level, the research relies on a comprehensive Integer Linear Programming (ILP) model that covers the entire Major Frame Allocation (MAF). This model aims to provide a holistic analysis of the system, ensuring that real-time constraints are consistently met throughout the system's operational cycle. The ILP model represents a strategic tool in evaluating system-level performance, particularly in complex multicore settings where multiple tasks and processes interact dynamically.

A pivotal aspect of this thesis is addressing the challenge of performing accurate Worst-Case Execution Time (WCET) analysis in the absence of Performance Monitoring Units (PMU) in the hardware. This situation is a common constraint in many industrial CRTES applications. This work demonstrates innovative methodologies that leverage software-based techniques to derive essential cache and bus event data necessary for WCET analysis. This approach is particularly vital for systems where hardware-based performance monitoring is not an option, providing a pathway to obtain critical performance data through alternative means.

The thesis, therefore, sets forth to not only critically analyze and build upon existing CRTES methodologies but to push the boundaries of current practices by introducing more precise, adaptable, and comprehensive approaches. These approaches are aimed at enhancing the predictability, reliability, and overall performance of CRTES in multicore environments, thereby addressing some of the most pressing challenges in the field, even under the constraints of limited hardware monitoring capabilities.

1.5 Contributions

The contributions of this Thesis can be summarized as follows:

1. **Comprehensive Review and Analysis**: A detailed review and analysis of the state-of-the-art in real-time multicore timing analysis, with a particular focus on inter-core contention modeling. This contribution identifies critical gaps in existing methodologies and sets the stage for the development of more robust and accurate

solutions for multicore systems.

2. **Iterative Method for Task-Level WCD Bounds**: Development of a novel iterative method to compute Worst-Case Contention Delay (WCD) bounds with task-level guarantees. This method advances existing approaches by offering a more precise and reliable prediction of contention effects, ensuring safe task execution in multicore environments. It is designed to balance accuracy and computational efficiency, making it suitable for both academic research and industrial applications. Moreover, this approach is fully automated, generating all necessary code based on system configurations, significantly simplifying its deployment.

3. **System-Level ILP-Based Approach**: Introduction of an ILP-based methodology that operates at the system level to derive tighter WCD bounds. By leveraging a global view of system resources and interactions, this approach reduces pessimism and provides more accurate guarantees for system-wide performance under worst-case conditions. Similar to the iterative approach, the ILP formulation is highly automated, with the capability to generate the required Python code for any system scenario. This automation enables practitioners to directly execute the ILP-based analysis with minimal manual effort.

4. **Development and Use of Microbenchmarks**: Design and implementation of microbenchmarks to fulfill multiple critical roles in this work. First, they are used to stress specific parts of the hardware, enabling controlled testing of contention scenarios. Second, they are tailored to generate high densities of particular hardware events, facilitating precise profiling of system behavior. Lastly, microbenchmarks are instrumental in validating the proposed methods. By running these benchmarks on other cores to simulate worst-case contention scenarios, it is possible to compare observed execution times with the derived WCET estimates, proving the safety and robustness of the proposed methods. This contribution underscores the importance of microbenchmarks as a tool for both experimental validation and system stress testing.

5. **Automated Implementation Framework**: Development of a fully automated framework to implement and execute both the iterative and ILP-based approaches. The automation includes not only the execution but also the generation of the Python code required for both methods based on system parameters, such as the number of cores and the set of events occurring on each core. This automation drastically reduces the time and effort required to apply the methodologies in practice, enhancing their usability and scalability.

6. **Proof-of-Concept on Real Hardware**: Execution of the proposed methodologies on real hardware to validate their performance and applicability. Using a widely adopted platform in the space industry, the experimental results demonstrate the practicality of these approaches, providing a bridge between theoretical models and

real-world applications. This validation confirms the feasibility and relevance of the proposed techniques for critical real-time systems.

7. **Software Solution for Non-PMU Systems**: Development of a software-based solution for WCET analysis on systems that lack a Performance Monitoring Unit (PMU). This solution represents a critical advancement for industrial settings where hardware monitoring tools may be unavailable, showcasing how precise timing analysis can still be achieved through careful software engineering and system-level insights.

8. **Integration of Advanced Analytical Techniques**: Incorporation of advanced analytical methods into the iterative and ILP approaches. These techniques enable a deeper understanding of multicore contention scenarios, modeling interleavings and overlaps with a high level of granularity. This ensures that the proposed methodologies are not only robust but also capable of adapting to the complexities of modern multicore systems.

1.6 Thesis Structure

The remainder of this thesis is structured as follows:

- **Chapter 1: Introduction.** This chapter provides an overview of the challenges in timing analysis for CRTES. It discusses the motivation and relevance of this work, the contributions made, and the structure of the thesis.

- **Chapter 2: State-of-the-Art.** This chapter reviews the existing methods and techniques in multicore timing analysis. It covers inter-core interference, contention management, and approaches to bounding contention delays, including the use of performance monitoring counters and statistical methods.

- **Chapter 3: System Model and Assumptions.** In this chapter, the hardware and system models are introduced. It describes the types of accesses, observability considerations, and the pairing of accesses in multicore systems. A cyclic executive assumption is also presented to contextualize the methods.

- **Chapter 4: Microbenchmarks.** This chapter details the design, implementation, and evaluation of microbenchmarks. These benchmarks are used for profiling and stressing the system. Challenges in implementing precise microbenchmarks and their relevance to timing analysis are also discussed.

- **Chapter 5: Design and Solution.** This chapter introduces two proposed methods for deriving WCETs: the iterative approach and the ILP-based approach. The chapter also includes architectural considerations and the rationale behind these solutions.

- **Chapter 6: Iterative Approach.** This chapter focuses on the iterative approach to compute the WCET. It discusses task-level guarantees, time composability, and the safety of the proposed method. Experimental results are provided to validate its applicability.
- **Chapter 7: ILP Approach.** This chapter presents the ILP-based approach for system-level WCET guarantees. It describes the constraints, objective function, and results of the approach. A proof-of-concept evaluation is also included.
- **Chapter 8: Profiling a Space Application on the GR712RC.** This chapter demonstrates the applicability of the proposed methods in an industrial scenario using the GR712RC platform. A software-based profiling solution is developed and validated. The results highlight the effectiveness of the approach for space applications.
- **Chapter 9: Conclusions.** The final chapter summarizes the contributions of the thesis, discusses the limitations of the proposed methods, and outlines possible extensions and future directions for this research.

1.7 List of Publications

1. **Accurate ILP-based Contention Modeling on Statically Scheduled Multicore Systems** - 25th IEEE Real-Time and Embedded Technology and Applications Symposium, Montreal, Canada, 2019. Authors: Xavier Palomo; Enrico Mezzetti; Jaume Abella; Reinder J. Bril; Francisco J. Cazorla. **Rate: Core A.** Link
2. **Tracing Hardware Monitors in the GR712RC Multicore Platform: Challenges and Lessons Learnt from a Space Case Study** - 32nd Euromicro Conference on Real-Time Systems, 2020. Authors: Xavier Palomo; Mikel Fernandez; Sylvain Girbal; Enrico Mezzetti; Jaume Abella; Francisco J. Cazorla; Laurent Rioux. **Rate: Core A.** Link
3. **ITER: An ITERative Approach for Inter-Core Timing Analysis in Statically Scheduled Cyclic Executive Systems on COTS Multicore Platforms for CRTES** - The Journal of Supercomputing, 2024. Authors: Authors: Xavier Palomo; Carlos Molina. **Rate: Q2.** Link
4. **Modeling Multicore Contention in Critical Real-Time Embedded Systems** - 9th URV Doctoral Workshop in Computer Science and Mathematics (DCSM), Tarragona, Spain, 2024. Author: Xavier Palomo.

Chapter 2

State-of-the-Art

The shift from single to multicore systems in critical embedded real-time domains has fundamentally altered the landscape of Verification and Validation (V&V), particularly in timing analysis. Over the past few decades, extensive research has been devoted to understanding the impact of hardware resource sharing in multicore systems [32], which has been clearly identified as the primary source of timing interference. In this context, this work does not consider interference arising from software resource sharing and inter-core synchronization protocols, which have already been addressed by existing analyzable protocols [16].

This chapter provides a comprehensive review of the scientific literature on inter-core timing interference caused by contention for shared hardware resources. We focus primarily on the shared bus, as it is the most common and extensively studied source of contention. The strategies developed in the literature can be broadly categorized into those that aim to precisely analyze the effects of contention and those that seek to control or eliminate it.

Understanding interference and contention delay is fundamental to this discussion. Interference in multicore systems occurs when the execution of tasks on one core affects the performance of tasks on another core due to competition for shared resources. Contention delay is the additional time required for tasks to wait for access to these shared resources. Accurately modeling this delay is essential for ensuring that real-time systems meet their stringent timing requirements.

Several methodologies have been proposed to mitigate contention effects. Hardware-based techniques, such as cache partitioning and dedicated memory channels, aim to physically isolate the resources used by different cores, thereby reducing contention. These techniques, however, often increase hardware complexity and costs. On the software side, intelligent task scheduling and dynamic priority assignment are employed to minimize resource access conflicts among tasks. These methods leverage scheduling algorithms that are aware of the tasks' resource usage patterns. Additionally, hybrid techniques combine hardware and software approaches to provide a balanced solution to managing contention. For instance, hardware monitoring data can inform software scheduling adjustments to optimize resource usage dynamically.

In addition to mitigating contention, accurately quantifying its effects is crucial. Ana-

lytical models use mathematical representations to predict contention delays based on detailed knowledge of hardware architecture and task characteristics. These models are essential for theoretical analysis and design. Simulation-based methods involve detailed simulations of task execution on multicore systems, providing insights into contention effects. These simulations can model various aspects of the hardware and software stack to produce accurate predictions. Empirical measurements, using performance monitoring counters (PMCs), gather data on resource usage and contention. This data can calibrate analytical models or validate simulation results. However, the absence of comprehensive PMCs in many industrial systems limits this approach [33].

Different scheduling assumptions significantly impact the effectiveness of contention analysis. Fixed-priority scheduling, where tasks have fixed priorities and higher-priority tasks can preempt lower-priority ones, is simpler to analyze but may lead to suboptimal resource utilization. Dynamic scheduling, where task priorities change at runtime based on factors such as task urgency and resource availability, improves performance but complicates analysis and implementation. Limiting resource usage by tasks through quotas can also help manage contention. For example, restricting memory access frequency ensures no single task monopolizes the resource.

PMCs play a crucial role in performance analysis for CRTES. They provide low-level data on events like cache misses and branch mispredictions, which are critical for detailed timing analysis. However, the lack of PMCs in many industrial systems poses a significant challenge [33].

The transition to multicore architectures in CRTES necessitates advanced techniques for timing analysis. This chapter outlines the state-of-the-art in contention analysis, highlighting the challenges and innovative solutions proposed in the literature. The subsequent sections will delve deeper into specific methodologies and their applications, providing a comprehensive overview of current approaches in real-time multicore timing analysis.

2.1 Inter-Core Interference and Contention Delay

To understand the challenges addressed in this work, it is crucial to differentiate between direct and indirect inter-core timing interference. Direct interference occurs when a task τ_i is delayed in accessing a shared hardware resource because another task τ_j, executing on a different core, is currently using that resource or has higher access priority. This type of interference specifically captures the contention effects during resource access. Throughout this work, unless otherwise specified, our focus will be on this direct form of interference, where the contention delay is solely produced by direct competition for resources.

Indirect interference, on the other hand, involves more complex interactions that do not stem directly from simultaneous resource access. For instance, a task running on one core might evict useful cache data from a shared cache, resulting in additional cache misses for another task when it tries to access that data. This type of interference can significantly impact performance in real-time systems [20]. Characterizing indirect interference precisely is challenging because it often requires making strong conservative assumptions to

ensure safety, which can lead to overly pessimistic timing predictions [6]. To mitigate such pessimism and better support the isolation requirements of modern mixed-criticality systems, various core-local resource partitioning techniques are employed. These techniques include both hardware and software mechanisms designed to ensure that critical tasks can execute with minimal unexpected interference.

The majority of research efforts aimed at bounding the effects of contention rely on the presence of core-local resources, such as L1 caches and scratchpads, or the use of segregation mechanisms like L2 cache partitioning. By leveraging these localized resources and mechanisms, it becomes feasible to exclude sources of indirect interference and more effectively control the direct contention effects. This segregation helps in providing a more predictable and manageable environment for critical tasks, thereby enhancing the overall reliability and performance of real-time systems.

Understanding and mitigating inter-core interference is essential for advancing the field of real-time embedded systems. Direct interference, while more straightforward to model, still requires precise characterization to ensure that timing constraints are met. Indirect interference adds a layer of complexity, necessitating advanced partitioning and isolation strategies to maintain system predictability. By focusing on both types of interference, this work aims to provide comprehensive solutions that enhance the predictability and performance of multicore real-time systems.

2.2 Controlling or Removing Contention Effects

Current research efforts demonstrate a strong trend towards minimizing or completely avoiding inter-core interference. A significant number of studies leverage the PRedictable Execution Model (PREM) [38], which organizes task execution into distinct read, execute, and write phases to minimize simultaneous bus accesses by different tasks. This model assumes that memory access instances can be predicted and isolated into specific phases, occurring in bursts rather than being evenly distributed throughout task execution. Typically, this involves loading the task's working set into local scratchpad memories and relies on some level of resource partitioning within the platform. When these assumptions are met, the contention effect can be mitigated by aligning the phases of different tasks so that read and write phases do not overlap across cores. Thus, a task performs its bus accesses during the computation phases of other tasks, and this model often works in tandem with task-to-core mapping strategies.

For instance, some authors attempt to avoid overlapping high-memory-exchange phases of tasks [39]. Blagodurov et al. [29] propose a task-to-core mapping algorithm based on heuristics, classifying tasks by their miss rates to prevent tasks with high cumulative miss rates from running concurrently. Similarly, another study [40] suggests an algorithm that uses bank partitioning to map memory-intensive tasks to the same core, thereby reducing contention by ensuring that each core accesses separate memory regions. Unlike these approaches, our work does not assume that phase isolation is always feasible, nor do we directly address task-to-core mapping, although our model could be adapted to support

these assumptions.

A specific scheduling framework for avionics multicore platforms is detailed in [41]. This framework characterizes tasks by identifying and isolating mandatory execution parts from optional or less critical ones, which can be delayed. It proposes a limited preemptive model using a Single Core Equivalence (SCE) technique to mitigate overhead. The SCE is achieved through a scheduler that supports temporal and spatial partitioning, enabling job skipping and allowing less critical task parts to have more relaxed performance requirements. Our approach does not use a preemptive model, thus avoiding the need to identify critical parts within tasks.

Another work [58] addresses parallel applications, assuming read-execute-write semantics, where task-to-core mapping and schedules are statically defined. It aims to introduce slack time in the schedule to reduce resource contention. This approach, while sharing similarities with our iterative method, tends to build on pessimistic task-level bounds and does not support pairwise mapping of different request types, unlike our ILP formulation, which avoids pessimism at the task level.

Focusing on DRAM and memory controller operations, the authors in [50] propose bank partitioning to bound the delay on memory accesses and reduce inter-core interference. While our work shares some similarities regarding access pairing, we differ in the memory-controller perspective and the minimum overhead approach, which considers both the task under analysis and interfering tasks.

While reducing contention is a reasonable goal, it can lead to reduced flexibility due to strong assumptions about application semantics. Typically, collision avoidance in task accesses relies on a phased task model, which is not always sustainable. Therefore, our research focuses on scenarios where such models are not possible, such as in legacy systems, or when quantifying the maximum potential contention delay is necessary, even if phased task models could theoretically be applied.

2.3 Accounting for the Worst-Case Contention Delay

Accurately accounting for the Worst-Case Contention Delay (WCD) in multicore systems is essential for ensuring that real-time tasks can meet their deadlines even under the worst interference conditions. Various approaches have been proposed to analyze and derive tight bounds on WCD, each operating under different assumptions and methodologies. This section will review some of these approaches, focusing particularly on those that align closely with the methodologies presented in this thesis.

Some approaches, such as the one in [24], operate exclusively at the task level. These methods calculate WCD by considering the worst possible alignment for each task, but they do not take into account the actual mapping of tasks to cores. This means they overlook the real or potentially feasible alignment of co-running tasks. Moreover, these approaches assume the worst-case scenario for every access request, considering the highest latency requests possible. This leads to overly pessimistic WCD calculations, as every running task on other cores is considered a potential contender. These conservative assumptions

result in a loose WCD bound, which we will use as a baseline for comparison in our experiments. Detailed comparisons are provided in Chapter 5.

Earlier research often focused on bounding the number of requests to a shared hardware resource for each task in isolation. For example, some works assumed a uniform distribution of accesses with a minimum distance between consecutive accesses [25], while others assumed dedicated task phases [26]. The upper bound of interference suffered by a task was determined by summing up the number of accesses from all potential co-runners. This interference bound could be tightened by considering only those co-runners that may actually overlap with the task under analysis. Studies such as [27] and [14] derived an upper bound on the effect of contention by analyzing the maximum number of bus requests issued from other cores within a given interval. This information was then used to compute the bound of contention effects on a task scheduled on another core, with the results integrated into standard response time analysis.

A common trait in these approaches is their primary focus on the activity of co-runners, rather than directly on the task under analysis. However, the approach in [14] considers both contender and contending tasks, although it does not assume multiple bus request types, leading to less precise WCD bounds. Each access is assumed to entail the longest latency possible. This method of computing the upper bound on the increase in execution time due to contention is similar to our iterative approach, but we also consider both the task under analysis and the requests of co-runners to derive tighter bounds. Our proposed methods will be compared to this approach to demonstrate how considering both tasks' bus requests can yield more accurate results.

Other approaches, such as [63] and [74], compute the WCD under the assumptions of the task phase. For example, [63] proposes a co-scheduling mechanism based on the PRedictable Execution Model (PREM) to manage contention at a high level. Similarly, [74] considers tasks as composed of superblocks, which are assigned to processing elements and executed in a time-triggered or sequential manner. These superblocks serve as dedicated phases, making them suitable for comparison with our methods.

2.3.1 Modeling the distributions of accesses to the shared resource

Beyond considering the number, type and latency of interfering and interfered accesses, it is crucial to understand the timing of these accesses. With detailed information on when accesses occur, more favorable task alignments and core mappings can be devised. This reduces the need to assume worst-case alignment scenarios, which often lead to overly pessimistic conflict assumptions.

However, obtaining precise timing information for resource accesses is practically infeasible and represents a significant source of pessimism in these models. Different approaches have been proposed to minimize this pessimism. As discussed in Section 2.2, some approaches rely on an execution model that divides task execution into memory and computation phases. Shared resource accesses occur exclusively during the memory phases.

This assumption is utilized in many works [64], [63], [73], [74], [39] to address task-to-core mapping more optimally compared to considering entire tasks.

Rosen et al. [15] measured cache miss effects on the bus, assuming that memory accesses occur only at the beginning and end of tasks. Tasks are scheduled to avoid overlapping memory phases. Other approaches assume that tasks' accesses to shared resources are evenly distributed over the task's execution [50]. However, this is rarely the case and is difficult to validate.

Although phased execution may be reasonable for parallel applications, it lacks generality and is incompatible with legacy code. Regardless of the assumptions about access distribution, the proposed approach is flexible enough to model different application characteristics and apply contention analysis at various granularity levels (i.e., full tasks, task sections, or phases). Resource requests can occur at any time during a task's execution, whether evenly distributed, sparsely, or in bursts.

2.3.2 Using ILP to bound contention effects

Integer Linear Programming (ILP) has been extensively explored in the research community for timing analysis, aiming to calculate optimal solutions given a set of constraints. This subsection explores relevant uses of ILP in WCD computation and task-to-core mapping, focusing on contention effects.

Some works use ILP to compute optimal task-to-core mappings to minimize contention, avoiding concurrent execution of memory-intensive phases of tasks. For example, [15] addresses non-independent runnables sharing data, using an ILP formulation to find the optimal timing for tasks to access memory, based on phased execution as in [7]. This approach seeks a contention-free environment through time-triggered scheduling with bank privatization, exploring optimal schedules for task memory accesses.

Other studies, such as [57], focus on parallel computing applications modeled as Directed Acyclic Graphs (DAGs) within a time-triggered model and a round-robin bus. They use ILP to optimally insert slack time between tasks, claiming improvements in memory-intensive scenarios. Our ILP formulation differs by not introducing slack time between tasks or influencing the timing of task accesses to shared resources.

Similarly, [72] proposes contention-aware time-triggered scheduling strategies, using knowledge of application structures to minimize contention and overall makespan. ILP is used to calculate optimal schedules, avoiding poor task alignments by setting trigger instants statically. While this approach shares similarities with our iterative approach, we do not seek optimal schedules and assume an already given task-to-core mapping.

Additionally, domain-specific platforms, particularly in the automotive sector, have seen ILP used to characterize contention effects. For instance, [25] presents an ILP model focusing on contention between task pairs but does not consider task overlapping. Our work, however, explores the circular dependence between contention and task alignment. In [49], ILP is used to derive WCD bounds in static time-triggered systems, modeling instruction caches, and assuming separate buses for data and instruction accesses, thus not considering their interference.

2.4 Contention Analysis and Scheduling Assumptions

Contention analysis methods can be further classified based on the scheduling model they operate under. Dynamically scheduled systems rely on real-time decision-making to manage tasks, whereas statically scheduled (or offline) systems depend on a predefined schedule established before execution begins.

In dynamically scheduled systems, the scheduler makes real-time decisions to ensure that critical tasks have sufficient resources to complete their execution. Several works [63], [64], [30], [22] focus on dynamic scheduling approaches. For example, Inam et al. implemented Behnam's Multi-Resource Server (MRS) [45], [13] for statically partitioned multicore real-time systems. This method considers additional sources of contention, such as CPU bandwidth competition among tasks, and employs a Fixed Priority Pre-emptive Scheduling (FPPS) policy for both node and server scheduling. Another approach [61] uses real-time monitoring to ensure that the requests of co-running tasks stay within a utilization threshold determined during analysis.

These dynamic scheduling methods differ from the scope of this thesis, as they primarily focus on real-time monitoring and the overhead caused by task preemption and context switching. The increase in execution time in these systems is often due to these overheads rather than direct contention effects.

In contrast, static (offline) scheduling aims to define a complete schedule before the system starts executing. This schedule is fixed and does not change during runtime, eliminating the need for real-time decision-making. Various studies [58], [42], [70] explore static scheduling methods. For instance, the work in [58] assumes preemptive scheduling without making work-conserving assumptions to preserve the analysis results at runtime. Another study [70] considers tasks as super-blocks that are executed sequentially or according to a time-triggered schedule. These paradigms, including the generic access model and dedicated phases (e.g., read-execute-write), are evaluated, with empirical evidence suggesting that dedicated phases result in better response times than treating tasks as monolithic units.

Additionally, it is important to distinguish between event-triggered and time-triggered tasks. Event-triggered tasks are activated sporadically in response to significant events [46]. Predicting the exact timing of these tasks is practically unfeasible, making dynamic scheduling more suitable for handling them [48], [49], [27], [14]. In most cases, event-triggered task scheduling involves task preemption, adding complexity to the scheduling process.

On the other hand, time-triggered tasks are activated at predefined intervals, making their timing predictable [50]. This predictability allows for cyclic scheduling, where tasks are executed in a regular, repeating pattern. In this work, we focus on scenarios where time-triggered tasks are executed cyclically, leveraging their predictable nature to develop more efficient scheduling algorithms.

Understanding the underlying scheduling assumptions is crucial for effective contention analysis. Dynamic scheduling provides flexibility and responsiveness, essential for

handling unpredictable task activations. However, it introduces overheads that can complicate timing analysis. Static scheduling, with its predetermined schedules, offers a more predictable environment, reducing the complexity of contention analysis. However, it may lack the flexibility needed for systems with highly variable workloads.

2.5 Enforcing Resource Usage Quotas

Regulating the allocation of hardware resources to cores has been extensively studied as a strategy to avoid or reduce conflicts arising from resource contention. Several approaches depend on specific assumptions about application semantics and hardware mechanisms, or they leverage Real-Time Operating System (RTOS) mechanisms to enforce predetermined resource usage quotas.

One such method is the implementation of memory bandwidth management systems, which enforce memory usage quotas. A notable example is presented in [71], where the authors differentiate memory bandwidth into two components: guaranteed and best-effort. In this approach, tasks that require guaranteed bandwidth are given priority, allowing them to run in temporal isolation. Meanwhile, tasks classified as best-effort share the remaining bandwidth and run concurrently, subject to contention. This separation ensures that critical tasks can execute without interference from lower-priority tasks, thereby improving the predictability of the system.

Another method for determining response times in multicore architectures involves extending the analysis techniques initially described by Tindell et al. [9]. This approach uses the concept of busy windows or intervals, focusing on the bus through which different components communicate and access memory. Unlike the methods proposed in this thesis, these analyses are designed for fixed priority preemptive systems. The emphasis on busy windows allows for a more detailed understanding of how contention affects task execution times, but it does not address the flexible scheduling requirements of many real-time systems.

When hardware isolation is not feasible, software-based mechanisms can be employed to manage resource contention. For instance, specific RTOS support, such as that provided by PikeOS [8], can be utilized to enforce utilization bounds determined during analysis. In this context, the authors of [52] propose a software thread-based mechanism that controls access to shared resources, thereby minimizing contention. However, managing resource quotas through software introduces additional overhead and may not always be feasible, particularly in systems without RTOS support.

In contrast to these methods, our approach does not impose any specific timing intervals during which tasks must access resources. This flexibility allows for more dynamic task scheduling and resource management, accommodating a wider range of application scenarios. By avoiding strict timing constraints, our approach can adapt to varying system conditions without sacrificing predictability or performance.

Despite the benefits of enforcing resource usage quotas, these methods often rely on strong assumptions about the system's behavior and capabilities. For example, memory

bandwidth management systems assume that tasks can be clearly categorized into guaranteed and best-effort phases, which may not always be practical. Similarly, software-based quota management requires support from the underlying RTOS, which may not be available in all systems.

Overall, while enforcing resource usage quotas can effectively reduce contention, it is important to consider the trade-offs involved. Hardware-based methods can offer high levels of isolation but at the cost of increased complexity and potential resource underutilization. Software-based methods, on the other hand, introduce overhead and dependency on RTOS features. Our approach aims to provide a balanced solution by avoiding rigid constraints and leveraging the inherent flexibility of dynamic scheduling and resource management.

2.6 Using Performance Monitoring Counters to Model Contention

Performance Monitoring Counters (PMCs) are a valuable tool for modeling contention in multicore systems. They provide detailed, low-level insights into how resources are being accessed and used, which is crucial for accurate contention analysis. In [25], the authors identify three essential factors needed to understand the effects of tasks accessing a shared resource:

- (i) The quantification of the amount of shared resource operations issued by a task and all tasks on a core.
- (ii) The analysis of the total latency experienced by a set of such operations on the shared resource.
- (iii) The inclusion of this delay in the response time analysis of each task.

We assume that this information can be derived, either directly or indirectly, from the PMCs available on Commercial Off-The-Shelf (COTS) platforms. Prior research [23], [25], [46] has employed PMC-based profiling to capture various aspects of resource access. For instance, Dasari et al. [23] used PMCs to monitor the number of accesses generated by each core in an event-triggered system. In their setup, caches were not shared to avoid variations in the number of requests due to interference. Although they did not implement access pairing (a technique we will discuss in Chapter 5), their solution shares some similarities with the iterative approach presented in this thesis.

In contrast, Dasari et al. [22] evaluated an iteration-based approach within a cyclic schedule for computing the maximum interference generated and suffered, ultimately deeming it unsafe. Their conclusion was based on the use of intervals to define the worst-case interference caused by a core. However, our approach maintains the integrity of the analysis assumptions by leveraging time-triggered activation and inter-core synchronization. Detailed examples supporting our methodology are provided in Chapter 5.

In the previous work [43], we introduced an innovative contention modeling technique tailored to a system model consistent with the assumptions presented in this study. The primary goal of that research was to derive precise bounds at the hypercycle level, offering an intricate understanding of system-wide contention dynamics. The current methodology, however, delves into a more granular level of analysis, focusing on determining the utmost upper-bound for each individual task. This is achieved by meticulously monitoring and identifying all potential co-running tasks that could coincide with a specific task under scrutiny.

Such detailed examination, while providing valuable insights, naturally induces a conservative outlook at the hypercycle level. Nonetheless, this approach is not merely academic; it holds practical relevance, particularly in scenarios requiring rigorous task-level assurances. When compared, the methodology from [43] and the one presented here serve as complementary tools, each addressing unique aspects of contention modeling in statically scheduled multicore systems.

Sections 3 and 5 will delve into the specifics of our methodology. We will explain the mechanisms by which we identify and account for all potential co-runners of a task under analysis. Furthermore, we will provide a comprehensive breakdown of our strategy to accurately compute the worst-case execution time (WCET) for each task. This systematic approach offers a deeper understanding of the task's runtime behavior in the presence of concurrent tasks on multicore platforms.

2.7 Statistical and Machine Learning Methods

Recent advancements in Worst-Case Execution Time (WCET) estimation are transforming the design approaches for real-time systems. A prominent direction in this field is Measurement-Based Timing Analysis (MBTA), as discussed in [9]. This study addresses the complexities introduced by multicore systems, instruction pipelines, branch prediction, and cache memory. By leveraging machine learning techniques, it establishes correlations between cycles per executed instruction and various hardware-related events, providing a practical alternative to traditional static timing analysis.

Another innovative approach is presented in [8], which introduces WE-HML, a hybrid WCET estimation technique. This method addresses the challenge of limited knowledge about processor microarchitecture by combining static analysis techniques with machine learning, operating directly on binary code for enhanced precision. The focus on cache modeling yields significant improvements in WCET estimates, particularly for ARM Cortex-A53 processors.

Furthermore, [52] tackles the critical need for early WCET determination during the initial stages of system development. By utilizing machine learning models at the source code level, this approach predicts WCET without relying on hardware specifics or compiled binaries. This method offers valuable early insights into WCET, which are crucial for real-time systems development. However, the accurate prediction of WCET at this early stage is challenging due to the lack of explicit knowledge about hardware specifics and the

absence of compiled binaries. In Critical Real-Time Embedded Systems (CRTES), where hardware-software interactions are vital, relying solely on source code may not capture the intricacies of the underlying hardware architecture.

In contrast, [75] explores Measurement-Based Probabilistic Timing Analysis (MBPTA) on a real multicore platform, considering the challenges posed by interference from the operating system and concurrent activities. Although Extreme Value Theory (EVT) is employed, the study highlights that EVT may not always yield adequate models and coherent probabilistic WCET (pWCET) estimations. This work raises awareness about the potential risks of applying MBPTA-EVT in real-world scenarios, emphasizing the need for careful implementation and verification of probabilistic WCET in safety-critical environments.

Addressing uncertainties in probabilistic WCET estimation, [69] introduces a region of acceptance model based on statistical test procedures. This model provides strategies for balancing safety and tightness in WCET estimation. Despite these improvements, the study acknowledges the inherent challenges and potential errors associated with probabilistic WCET analyses, especially in safety-critical environments. Both works contribute valuable insights into the complexities of probabilistic WCET estimation, underscoring the importance of meticulous consideration and validation when applying these methods to real-world embedded systems.

However, both probabilistic and machine learning approaches face significant challenges in CRTES. The need for accurate, tight, and deterministic WCET estimates is paramount to ensure system safety and meet stringent timing constraints. Uncertainties and the trade-offs between safety and precision make these methods less viable for critical real-time domains. While they are suitable for obtaining early-stage WCET estimates, their application in CRTES is limited due to the potential for errors and the critical need for precise timing guarantees.

Chapter 3

System Model and Assumptions

In the domain of real-time embedded systems, particularly those utilizing multicore architectures, deriving reliable and tight upper bounds for multicore contention is a challenging task. The extent of contention a task might endure is intrinsically linked to the specificities of the system's hardware and software configurations. Broad, generalized formulations for estimating this contention often lead to excessive pessimism, diminishing the practical applicability of the results. Therefore, a nuanced understanding of both system-level (software) and hardware-level factors is essential for an accurate analysis of contention effects.

This thesis addresses the complexities associated with multicore contention by developing a comprehensive approach that is grounded in a detailed understanding of both software and hardware aspects of multicore systems. We begin by outlining a generic system model that captures the essential elements common to various multicore platforms. This model encompasses fundamental aspects of multicore processing, such as shared resource access, scheduling mechanisms, and task execution dynamics, which are critical to contention analysis.

Shared Resource Access

A key aspect of our system model is how tasks access and contend for shared resources, such as memory buses and caches. The model addresses the impact of these shared resources on task execution times and overall system performance. Understanding the dynamics of how multiple cores interact with shared resources is crucial for predicting and mitigating contention effects.

Scheduling Mechanisms

The scheduling approach, especially in real-time systems, significantly influences system behavior. Our model considers both static and dynamic scheduling mechanisms, focusing on their implications for task-level predictability and system-level performance. Different scheduling policies can either exacerbate or alleviate contention, depending on how they allocate resources and order task executions.

Task Execution Dynamics

Understanding the dynamics of task execution, including task interactions and their influence on overall system timing, is integral to our model. This includes considerations of task priorities, execution order, and potential delays due to contention. The interplay between task scheduling and resource contention must be carefully modeled to ensure accurate timing predictions.

In subsequent chapters, we delve into the specifics of how these generic concepts manifest in different architectures. Each architecture presents unique challenges and characteristics that necessitate tailored approaches to contention analysis and timing prediction. The granularity and specificity of our analyses reflect the distinct architectural features and operational contexts of these systems, ensuring that our conclusions and recommendations are both accurate and relevant.

3.1 Hardware Model and Observability

3.1.1 Generalized Hardware Model

In multicore systems used in Critical Real-Time Embedded Systems, the hardware architecture significantly influences system performance and predictability. Understanding the underlying hardware model is essential for accurate analysis and modeling of contention and timing interference.

Contention in Shared Resources

Multicore systems typically feature several shared hardware components, such as memory buses, caches, and I/O interfaces. Contention arises when multiple cores concurrently access these shared resources, leading to potential delays and timing interference. This interference is cumulative across all accessed shared resources. For instance, in some COTS platforms like the Infineon AURIX Tricore families, contention may occur during access to flash memories, static RAM, or interconnect interfaces.

Role of the Interconnect

A critical component in these systems is the interconnect, which often serves as a primary source of contention. In configurations where the interconnect can handle multiple requests simultaneously, contention shifts from the interconnect to the request destinations. However, in systems where all external (off-chip) requests traverse a shared bus capable of serving only one request at a time, the bus becomes the primary contention point. The arbitration protocol of the bus, such as round-robin, plays a significant role in managing this contention.

Cache Architecture and Policies

Our generalized hardware model includes cores with private L1 instruction and data caches, accessing shared resources like main memory or a shared L2 unified cache through a shared bus. When an L2 cache is present and shared among the cores, we assume cache partitioning to minimize indirect inter-core interference. Typical cache policies in such systems include write-through no-write-allocate for L1 caches and write-back for L2 caches or off-chip SRAMs. These policies, commonly found in embedded COTS platforms, significantly influence the system's behavior under contention.

3.1.2 Observability and Access Types

Measuring Contention

A key aspect of analyzing multicore systems is the ability to observe and measure the effects of contention. The time taken to serve a request and the consequent latency depend on the type of request being made. This variability is crucial for understanding the system's performance under different load conditions. To effectively analyze and model contention, we assume the capability to determine the pattern of requests made by each task. This information, encompassing the number and types of accesses, can typically be derived through two primary methods:

Static Analysis: This method involves analyzing the code of the task to predict its behavior, including access patterns. Static analysis tools and techniques provide a way to estimate the request patterns without executing the task.

Dynamic Profiling: Alternatively, dynamic methods involve monitoring the task's behavior during execution. This approach often relies on Performance Monitoring Counters, available in the Performance Monitoring Units of modern COTS platforms. Dynamic profiling allows for the collection of real-time data on the task's access types and frequency.

Request Type and Latency

In our model, we recognize that different types of accesses (e.g., memory read/write operations) will incur varying latencies. The type of access significantly impacts the extent of contention experienced by a task. Understanding these nuances is vital for developing accurate models of contention and for implementing effective strategies to mitigate its impact. Each type of access identified in a task's pattern is associated with its worst-case latency. This latency is dependent on the specific architecture of the target platform. For practical applicability, our approaches do not mandate the use of a specific analytical method, like data flow analysis, to derive these latencies. Instead, we assume that the required information is obtainable either via hardware (such as PMCs) or via software profiling tools. This approach allows tasks to be dynamically profiled, capturing accurate data on access types and their respective counts.

Applicability of the Model

It is important to note that the focus on systems with a shared bus does not limit the applicability of our methodologies. The proposed approaches are versatile and can be adapted to different system configurations, including those with more complex interconnect architectures or different cache policies.

In subsequent chapters, we will assume the presence of observability mechanisms. However, we will also offer solutions for scenarios where such mechanisms are not available, ensuring that our methods remain broadly applicable across various multicore system architectures. These solutions will address the unique challenges posed by different hardware and software configurations, demonstrating the flexibility and robustness of our contention analysis approaches.

3.2 System Model

The Thesis encompasses a collection of m discrete periodic tasks subject to constrained deadlines, symbolized as $\mathcal{T} = \tau_1, \tau_2, \ldots, \tau_m$. Each individual task τ_i can be characterized by a quartet (c_i, r_i, P_i, D_i), where c_i and r_i are respectively indicative of the worst-case execution time and the earliest release time of τ_i, while P_i and D_i outline the task's repeating period and its relative deadline. The primary emphasis of this work lies in the exploration of contention effects, excluding the quest for an optimal scheduling or core task assignment [72]. As such, a static assignment of tasks to cores is presumed, and migration of tasks across cores is not considered. A uniform set of cores, denoted as Π, is contemplated, and for each core within the system, $\mathcal{T}_s$ is employed to single out the specific subset of $\mathcal{T}$ allocated to a particular core π_s, defined as $\mathcal{T}_s = \tau_i \in \mathcal{T} \wedge \tau_i \mapsto \pi_s$. Additionally, the scheduling of tasks on each core is orchestrated non-preemptively, abiding by an implicit sequence that mirrors the underlying precedence restrictions between tasks. The scheduling mechanism assumed here is not work-conserving, meaning that a job will remain idle until the conclusion of the reserved time frame (or timing budget) corresponding to the previous task's job, even if completion has already been achieved. Lastly, it is taken for granted that inter-core dependencies amongst tasks are either nonexistent or do not influence the sequential execution order of tasks.

3.3 Types of Accesses

In the context of a homogeneous multicore processor system, the nature and targets of accesses remain uniform throughout the structure. The specifics of an off-chip access target might carry significance if multiple interconnects are present or if the interconnects offer some level of parallel processing capabilities. Nevertheless, to ease the notation's complexity, targets of operations can be integrated within the operation type itself (for instance, the operation type could be categorized as reading or writing to a particular target). Specific to our reference architecture, there is no necessity to distinguish the target of

a request since all requests are routed through a singular shared bus. Consequently, target details have been excluded from the model's formulation. Various requests, symbolized as $t^1, \cdots, t^k \ni \Gamma$, are processed via the interconnect, and each request $t \in \Gamma$ is characterized by a distinct latency. An upper boundary for this latency, denoted l^t, is ascertained (for instance, through comprehensive testing). The notation a_i^t is utilized to pinpoint the set of access types t executed by task τ_i. Accordingly, a_i represents the total set of accesses, irrespective of their type, carried out by that task.

The particular kinds of accesses that may be taken into account, as well as their corresponding latencies, are dependent on the platform. Details concerning various access types and latencies will be acquired and evaluated on a specified platform, as will be demonstrated in Section **??**. Table 3.1 encapsulates the notations pertinent to the task model, various access types, and latencies, adhering to the conventions to be followed throughout this Thesis.

Tasks, cores and mapping	
$\Pi \stackrel{\text{def}}{=} \{\pi_0, \cdots, \pi_{n-1}\}$	Considered set of homogeneous cores
$\mathcal{T} \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_m\}$	Considered task set
$P_i \geq D_i$	Period (P_i) and deadline (D_i) of task τ_i
c_i	Worst-case execution time in isolation of τ_i
r_i	Release instant of task τ_i
$\mathcal{T}_s \stackrel{\text{def}}{=} \{\tau_i : \tau_i \mapsto \pi_s\}$	Subset of tasks statically mapped to core π_s
$\Gamma \stackrel{\text{def}}{=} \{t^1, \cdots, t^k\}$	Access types admitted in the system
l^t	Upper bound to the latency incurred by a single access of type $t \in \Gamma$
$l^{\max}$	Worst-case latency incurred by a single access of any type in $t \in \Gamma$
a_i^t	Number of accesses of type t in τ_i
$a_i \stackrel{\text{def}}{=} \sum_{t \in \Gamma} a_i^t$	Total number of accesses issued by task τ_i

Table 3.1: Task and core mapping notation used throughout the Thesis

3.4 Cyclic Executive Assumption

Real-time systems can be broadly categorized into event-driven or time-driven, based on whether actions are triggered in response to specific events or at designated time intervals. In this work, we align with the latter paradigm, assuming that tasks are executed non-preemptively according to a time-triggered approach [51], often referred to as cyclic executive scheduling. This method is prevalent in critical embedded real-time systems, attributable to its inherent predictability and minimal implementation requirements.

The underlying principle of cyclic executive scheduling involves cycling through a statically predetermined, recurring sequence of activities at a fixed frequency. Given that the initiation times for each job are calculated statically (offline) and that tasks are non-preemptive, there is no actual need for RTOS support for scheduling.

The entire set of tasks or activities is fully executed within a single hyperperiod, further segmented into smaller frames. Timing deadlines are imposed exclusively at the boundaries of these frames. In other words, every task must be executed and confined within an individual frame. While a frame might comprise multiple tasks, it can also include segmented portions of tasks, offering opportunities for optimal frame allocation.

Within each frame, lower-priority sporadic activities may optionally be executed during the available slack slots, which are not occupied by periodic tasks. Typically, the scheduling of non-periodic work must be preemptive to ensure the timely commencement of the subsequent frame [1].

Logically, the cumulative execution times (including any interference) of all tasks assigned to a frame must be less than or equal to the time slot allocated for that frame. To preclude the possibility of frame overruns, it is imperative that non-preemptive tasks conclude their execution within the designated frame. This can be accomplished through the application of precedence constraints (such as priority assignment), compelling task τ_{i-1} to complete prior to the initiation of task τ_i, facilitated by the orchestrated release times of the tasks.

Task	f [Hz]	T [ms]
A	40	25
B	20	50
C	10	100

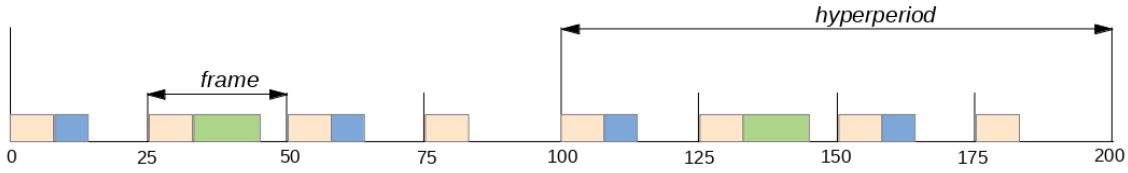


Figure 3.1: Generic example of cyclic executive scheduling

Figure 3.1 delineates the essential principles of cyclic executive systems. To derive an appropriate schedule, an offline computation phase is indispensable. For the context of this Thesis, the offline phase encompasses the calculation of the Worst-Case Delay (WCD) for each task, serving as a means to furnish secure frame-level timing assurances.

It merits emphasis that the magnitude of contention, as well as the probability of simultaneous contending requests, is predominantly governed by the system architecture. In this work, we scrutinize a multicore instantiation of the ARINC 653 framework, wherein a static scheduler orchestrates the recurrent execution of a major frame (MAF)—analogous

to the hyperperiod—subsequently partitioned into a series of minor frames (MIF). Within each MIF, a statically allocated ensemble of periodic tasks is executed non-preemptively atop a multicore platform.

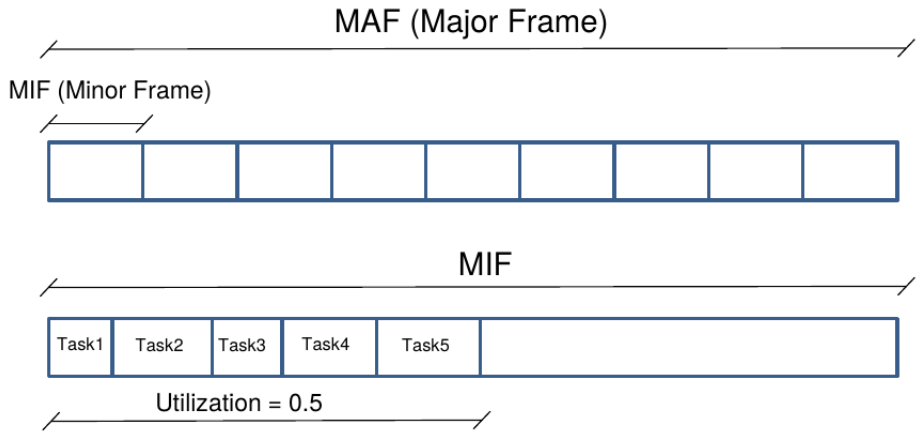


Figure 3.2: Task budgets within the MIF and MAF concepts.

Figure 3.2 depicts the mapping of tasks to a specific MIF, where each task is allocated a slot corresponding to the pre-computed timing budget. Achieving high utilization at the MIF level (by allocating more tasks) is desirable, but this goal must be balanced with the need for safe, reliable timing guarantees. These guarantees ensure that the MIF deadline will not be overrun under any execution conditions or critical alignment of possible events, even when considering the effects of contention.

In the example shown in Figure 3.2, a 50% utilization of the MIF would indicate that the timing budget, computed based on the isolated execution times of the mapped tasks, only covers 50% of the MIF. It leaves an open question as to how much of the remaining 50% can be utilized to accommodate additional tasks, or whether it will be sufficient to absorb the effects of contention. Providing a precise answer to this question is the ultimate objective of our work.

This work draws on the concepts of Integrated Modular Avionics (IMA) and the ARINC-653 standard [10], although the underlying reasoning can be generalized to encompass statically scheduled systems at large. In such systems, tasks or functions are executed within predetermined scheduling slots, each of a specific duration. Within the context of this work, MIFs are considered the smallest time intervals for which timing guarantees must be ensured. Thus, the static schedule is characterized by a fixed allocation of tasks to MIFs, which are then executed sequentially within the MAF.

In this multicore setting, each core's computational resources are allocated following the same static pattern (as defined by the MIFs and MAF), leading to synchronization between cores at the MIF boundaries. This kind of core synchronization at MIF boundaries is exemplified in the IMA multicore implementation supported by the SYSGO PikeOS Real-Time Operating System (RTOS) [5], as well as in the Multicore Multiple Independent Levels of Security/Safety (MILS) paradigm endorsed by VxWorks RTOS [6, 62]. Even

though functions on different cores may share the same scheduling slots, they do not participate in any synchronization or communication protocol, maintaining time-composability, as per the principles laid out in [14].

3.5 Contention and Pairing of Accesses

Contention occurs when multiple requests are directed towards the same shared resource. In our approaches, contention is modeled using the concept of *paired* accesses between tasks running on separate cores. In the reference platforms under consideration, the bus serves as the primary source of contention, with pending requests being handled according to a well-defined policy (in this case, a round robin strategy). Owing to the implementation of the round robin arbitration policy, a given access may be paired with at most $|\Pi - 1|$ other accesses (where Π symbolizes the set of cores in the system). This pairing can occur with accesses from different cores, irrespective of the nature of these accesses, which will ultimately determine the ensuing delay. The collective number of accesses a_i for a task τ_i is the cumulative total over all specific access types associated with that task, as well as store and load hits. To incorporate the impact of contention, the budget allocated to a task is expanded by adding the (worst-case) latency corresponding to each paired access, as dictated by the access type of the competing requests. Therefore, it becomes necessary to devise a model that accounts for pairings while also differentiating among the various access types: for each task τ_i, the number of dispatched accesses a_i^t for every access type t must be considered. It is crucial to consistently reference the worst-case access counts, which must be valid across all possible paths in the program, as justified in [21].
A task might experience an elongation in its execution time as a result of contention for a specific shared resource. Two scenarios may lead to a delay in a task attempting to access a resource: either two or more tasks try to utilize the shared resource at the exact same moment, or a task attempts to do so while the resource is already engaged by another task. While in the first scenario the incurred contention will be l^t, in the second scenario it may only constitute a fraction of l^t. Nevertheless, given the virtual impossibility of precisely forecasting the moments at which tasks will initiate their accesses, a conservative approach necessitates assuming a latency of l^t whenever an access is *paired*.

Deriving safe and tight bounds to the WCD is crucial for the computation and apportionment of reliable timing budgets to the functions provided by the system. COTS multicores feature several shared hardware resources, which in turn represent just as many sources of contention. While all sources of contention are relevant for deriving trustworthy budgets, in the scope of this work we focus on the problem of computing reliable bounds to the WCD potentially suffered by a given program or task when accessing the off-chip memory.

Accesses to the off-chip memory are one of the most critical sources of interference [73, 22, 25]. Every access to the memory hierarchy (either direct or in response to a cache miss) must pass through the interconnect, which easily becomes a performance bottleneck and, equally important, a huge source of timing variability. The worst-case

contention delay suffered by a task on bus access is a function of both (i) the number of accesses the task itself performs, and (ii) the number and type of accesses performed by its co-runner tasks [21, 47, 25]. Observation (i) is simply dictated by the fact that, based on conservative assumptions on access alignment and considering the specific arbitration policy on the bus, a task can suffer a contention delay at every single access in the worst case. However, as per observation (ii), the WCD is also determined by the set of potential co-runners of a task: first, the number of contending accesses cannot be larger than the number of accesses of other tasks running in parallel; second, the WCD depends on the type of accesses performed by the contender as not all accesses exhibit the same latency.

In this chapter we first introduce access pairing as the core technique we build on for the computation of conflicting accesses on a given hardware resources. Based on this, we present two approaches for computing the WCD at either task or core level.

Chapter 4

Microbenchmarks for Multicore Timing Analysis

4.1 Characteristics of Microbenchmarks

A defining characteristic of microbenchmarks is their narrow scope and intentional design around a single event, operation, or subsystem behavior. Unlike comprehensive benchmarks that measure the interplay of multiple components, microbenchmarks target a specific hardware or software aspect—such as forcing particular cache misses, stressing a memory bus, or exercising a single instruction repeatedly. By maintaining a strict focus, microbenchmarks enable precise measurement and fine-grained analysis of the event of interest.

This deliberate purpose-driven approach ensures that the microbenchmark is crafted to provoke the desired hardware or software response. For example, a microbenchmark might be designed to highlight the performance implications of L2 cache misses by accessing memory regions deliberately larger than the cache capacity. This event-focused strategy reduces the risk of introducing confounding factors and provides a clear, reliable measurement of how individual system components behave under controlled conditions.

Event Isolation

A critical advantage of microbenchmarks lies in their ability to isolate specific events from the complexity of the full system. Real-world applications often involve numerous concurrent activities, making it challenging to determine the exact source of observed performance effects. By stripping away unrelated operations and focusing on a single event, microbenchmarks yield measurements that directly attribute performance costs to that event alone. This isolation is essential for identifying subtle performance bottlenecks, validating performance counter readings, and refining architectural models without the noise and interference inherent in more complex workloads.

Customizability and Flexibility

Microbenchmarks are inherently adaptable. Their focused and modular design allows them to be tuned to various hardware configurations, system states, and performance objectives. Parameters such as data set size, iteration counts, and memory access patterns can be easily adjusted to explore a wide range of scenarios. This flexibility is particularly valuable in multicore systems, where tailored microbenchmarks can reveal how specific architectural features, shared resources, or contention scenarios influence overall system performance.

Event-Specific Microbenchmarks

Given the need for precision and isolation, microbenchmarks are often event-specific. Different microbenchmarks are designed for each type of event or behavior that needs to be measured. For example, one microbenchmark might measure L2 cache load misses, while another focuses on store operations or branch mispredictions. This event-specific approach allows for detailed analysis of how different system components and behaviors contribute to overall performance.

This specificity is especially important in multicore systems, where interactions between cores and shared resources can significantly impact performance. Microbenchmarks enable targeted analysis of these interactions, revealing how specific events on one core may affect another or how shared resources behave under different workloads.

4.2 Relevance in Profiling and Timing Analysis

As discussed in previous chapters, profiling and timing analysis are crucial processes in the development and verification of real-time systems, particularly in environments where timing guarantees are essential, such as aerospace, automotive, and industrial automation. The accuracy of these analyses directly impacts the system's ability to meet stringent timing requirements and ensure reliable operation under all conditions. Within this context, microbenchmarks are one of the necessary pieces that provide precise, targeted insights into the performance characteristics of specific system components or operations, thereby enhancing the overall effectiveness of profiling and timing analysis.

Profiling refers to the process of collecting data about the execution of a program, such as the frequency of specific instructions, cache hit/miss ratios, or memory access patterns. Regardless of the concrete methodologies used afterwards, this data is invaluable for understanding how a system behaves under different workloads and for identifying potential performance bottlenecks. Timing analysis, on the other hand, involves predicting or measuring the WCET of tasks within a system. This is particularly important in real-time systems, where tasks must complete within predefined time bounds to ensure system stability and safety.

Enhancing Profiling Accuracy with Microbenchmarks

Microbenchmarks contribute significantly to the accuracy and reliability of profiling data. By isolating specific events or behaviors, microbenchmarks enable precise measurement of how these events impact system performance. For example, a microbenchmark designed to generate a specific number of L2 cache misses can be used to verify the accuracy of performance monitoring counters by comparing the expected number of misses with the number recorded during profiling. This validation process ensures that the PMCs are correctly capturing the events they are designed to monitor, which is critical for accurate profiling.

The ability to target specific events means that microbenchmarks can be used to calibrate and fine-tune the profiling process. For instance, if profiling reveals unexpected performance characteristics, microbenchmarks can be deployed to investigate these anomalies in detail. By providing a controlled environment where only the event of interest is varied, microbenchmarks help to isolate the root cause of performance issues, allowing for more informed decisions about system optimization.

Moreover, microbenchmarks can be used to stress-test the profiling infrastructure itself. In scenarios where profiling tools are integrated into the system, it is essential to ensure that these tools do not introduce significant overhead or interfere with normal system operation. Microbenchmarks can simulate high-intensity workloads that push the profiling tools to their limits, revealing any potential weaknesses or inaccuracies in the profiling data.

Supporting Timing Analysis with Precise Event Data

In the context of timing analysis, microbenchmarks provide the detailed event data needed to model and predict worst-case execution times accurately. Timing analysis often relies on understanding how specific events, such as cache misses or memory access delays, contribute to the overall execution time of a task. Microbenchmarks are uniquely suited to generating the specific event data required for this analysis.

For instance, consider a timing analysis scenario where the goal is to determine the WCET of a task running on a multicore processor. In such systems, shared resources like caches and memory buses can introduce variability in execution times due to contention between tasks running on different cores. Microbenchmarks designed to simulate high levels of contention can be used to measure the impact of these shared resources on execution time. By running these microbenchmarks in conjunction with the task under analysis, it is possible to observe how the task's execution time changes under different contention scenarios.

This approach is particularly valuable in systems where the availability of hardware PMUs is limited or nonexistent. In such cases, microbenchmarks can be used to generate the necessary event data that would otherwise be collected by PMUs. For example, in a system without a PMU, a microbenchmark might be used to generate a known number of memory accesses, which can then be observed using alternative tracing mechanisms. The

data collected from these traces can be used to infer the behavior of the system under different conditions, providing the information needed for accurate timing analysis.

Microbenchmarks also play a critical role in validating the assumptions made during timing analysis. For instance, when deriving worst-case delay bounds, it is often necessary to assume that certain events will occur with a particular frequency or that specific patterns of behavior will manifest under certain conditions. Microbenchmarks can be designed to test these assumptions directly, ensuring that the timing analysis models are based on realistic and validated data. This validation process helps to reduce the conservatism often associated with worst-case timing estimates, leading to more accurate and usable timing bounds.

Microbenchmarks in Multicore Systems

The relevance of microbenchmarks becomes even more pronounced in multicore systems, where the complexity of interactions between tasks and shared resources can make profiling and timing analysis particularly challenging. In multicore systems, tasks running on different cores can interfere with each other in unpredictable ways, especially when they compete for access to shared resources like caches, memory buses, or I/O systems.

Microbenchmarks are essential tools for understanding and quantifying this interference. By designing microbenchmarks that specifically target shared resources, it is possible to measure the degree of contention that occurs under different operating conditions. This information can then be used to inform both the profiling process and the timing analysis, ensuring that the models used to predict system behavior are based on accurate and context-specific data.

For example, in a multicore system where two tasks are expected to run concurrently, a microbenchmark could be used to simulate a worst-case scenario where both tasks are accessing the same memory bank simultaneously. The resulting data would provide insight into the potential delays caused by this contention, which could then be factored into the timing analysis to produce a more accurate WCET estimate.

Moreover, in systems where the exact behavior of shared resources is difficult to model analytically, microbenchmarks offer a practical alternative. By empirically measuring the performance impact of specific interactions, microbenchmarks provide real-world data that can be used to validate or refine analytical models. This empirical approach is particularly valuable in complex systems where the theoretical models may be overly conservative or fail to capture all the nuances of system behavior.

Ensuring Reproducibility and Consistency

Reproducibility is a key concern in both profiling and timing analysis. The ability to reproduce results across different runs, different systems, or different configurations is essential for ensuring the reliability of the analysis. Microbenchmarks contribute to reproducibility by providing a controlled and well-defined workload that can be consistently applied across different environments.

By using microbenchmarks to generate the profiling and timing data, researchers and engineers can ensure that the conditions under which the data was collected are clearly defined and can be replicated. This is particularly important in industrial settings, where the results of profiling and timing analysis may be used to make critical decisions about system design, optimization, or certification.

4.3 Implementation

The implementation of microbenchmarks translates their conceptual principles—focus, purpose-driven design, event isolation, and flexibility—into concrete strategies and coding practices. While the underlying concepts are introduced in Section 4.1, the emphasis here is on practical considerations and methodologies for turning these ideas into actionable, measurable code.

4.3.1 Design Considerations for Microbenchmarks

Designing and implementing microbenchmarks involves carefully translating theoretical objectives into concrete artifacts that reliably generate the intended events. The following considerations guide this process:

- **Targeting Specific Events:** Building on the focused nature of microbenchmarks, implementation details must ensure that the generated code path consistently produces the intended event. For example, accessing an array sized to exceed the L2 cache capacity triggers repeated cache misses. Low-level control structures (e.g., assembly instructions or carefully tuned memory layouts) help maintain a high density of the targeted operation.

- **Minimizing Confounding Factors:** Reducing extraneous operations—such as branching, complex arithmetic, or I/O—ensures that the measured event dominates the performance profile. Techniques like loop unrolling or careful alignment of data structures help maintain a tightly controlled environment with minimal noise.

- **Hardware-Aware Tuning:** Architectural properties, including cache line sizes, memory hierarchies, and instruction pipelines, influence benchmark behavior. Aligning data structures, priming caches, and iteratively tuning memory-access patterns ensure that the microbenchmark accurately reflects steady-state performance conditions relevant to the targeted event.

- **Flexibility and Parameterization:** To support various platforms and research objectives, microbenchmarks are built with configurable parameters. Adjusting data sizes, iteration counts, or access strides provides a wide experimental space, accommodating different system architectures or testing scenarios.

- **Validation and Calibration:** Finally, cross-checking observed results with expected outcomes validates both the microbenchmark and the underlying profiling methodology. By designing microbenchmarks where the quantity of operations is precisely known, the resulting performance data can confirm that profiling tools are capturing events correctly and that timing analysis aligns with measured behaviors.

This implementation-focused perspective complements the conceptual characterization of microbenchmarks, ensuring that the abstract principles outlined earlier are concretely realized in code. Through careful tuning, parameterization, and validation, microbenchmarks serve as powerful tools for profiling, timing analysis, and system-level validation in multicore architectures.

4.3.2 Implementation Details for Different Events

The implementation of microbenchmarks for targeted events applies the principles discussed earlier—such as event isolation, hardware-aware tuning, and minimal overhead—to a concrete algorithmic structure. As shown in Algorithm 1, the process is divided into two phases: *Initialization* and *Execution*. This structured approach ensures that each event of interest, whether cache misses, bus contention, or pipeline stalls, is induced reliably and measured precisely.

Algorithm 1 Microbenchmark Structure with Initialization and Execution Phases

```
1: procedure INIT_UB
2:     // Data Structure Preparation
3:     Create necessary data structures (e.g., linked lists, arrays)
4:     Align data structures in memory for efficient access
5:     // Cache Initialization
6:     One iteration of RUN_UB to avoid future cold misses
7: end procedure
8: procedure RUN_UB(ITER)
9:     for i = 0 to ITER − 1 do
10:        Perform OPERATION 1
11:        Perform OPERATION 2
12:        Perform OPERATION 3
13:        ⋮
14:        Perform OPERATION N
15:    end for
16: end procedure
```

Algorithm Overview

Algorithm 1 implements the key design principles introduced in previous sections. The `init_ub` procedure ensures that the environment is stable before measurements begin. This

includes warming up caches and pipelines so that observed performance does not reflect transient start-up conditions. The `run_ub` procedure then executes a tightly controlled loop of operations that trigger the desired hardware events.

Assembly Implementation for Event Precision

To ensure that the workload precisely matches the intended event patterns, the microbenchmark operations are written in assembly. This low-level control avoids unintended compiler optimizations or memory access rearrangements that could dilute the targeted event density. For example, a benchmark inducing L2 cache misses can directly address memory locations chosen to be out of cache range, ensuring a predictable and steady miss stream.

Initialization Phase

In the initialization phase, previously discussed concepts are put into practice:

1. **Data Structure Preparation:** Allocation and alignment steps ensure that memory is accessed predictably, minimizing confounding factors such as misaligned cache lines.

2. **Cache and Pipeline Stabilization:** By performing a preliminary run of the benchmark operations, caches and pipelines reach steady-state conditions. This step ensures that measurements reflect stable operating scenarios rather than transient start-up effects.

3. **Performance Counter Setup:** For platforms with PMUs or other monitoring capabilities, counters are initialized here. On platforms without such units, alternative debugging registers or tracing methods are readied for data collection.

This phase translates the abstract principle of "controlled conditions" into tangible steps that ensure reproducible, noise-free measurement environments.

Execution Phase and Loop Unrolling

The execution phase (`run_ub`) applies loop unrolling and straightforward, repetitive operation patterns. These implementation details, introduced conceptually before, now ensure:

1. **High Operation Density:** By reducing branching overhead, the unrolled loop maintains a steady flow of the targeted operations (e.g., memory loads or stores), maximizing event generation.

2. **Consistent Access Patterns:** Unrolled loops enable better alignment with cache line boundaries and maintain a predictable memory access pattern, ensuring the integrity of the measured events.

Tailoring Benchmarks for Specific Events

With the algorithm's foundational elements in place, tailoring microbenchmarks to specific events becomes a matter of selecting appropriate operations and data structures:

- **Cache Miss Benchmarks:** Arrange memory accesses to continuously step through regions exceeding the cache's capacity, guaranteeing consistent misses.
- **Bus Contention Scenarios:** Execute multiple concurrent or burst memory operations to induce congestion and reveal how the bus behaves under load.
- **Pipeline Stall Conditions:** Craft sequences of dependent instructions to induce deliberate stalls, elucidating how pipeline dependencies translate into execution delays.

These event-specific adjustments reflect earlier described flexibility and focus, turning conceptual performance targets into reproducible, measurable phenomena.

Validation through known Outcomes

Finally, microbenchmarks serve as an internal consistency check. By constructing scenarios with known operation counts or predicted event frequencies, the profiling and timing analysis tools can be verified against the microbenchmark's results. This closed-loop validation ensures the reliability of both the measurement methodology and the microbenchmarks themselves, providing confidence in the quality of subsequent performance analysis work.

4.3.3 Challenges and Solutions in Implementing Microbenchmarks

Implementing microbenchmarks in multicore timing analysis involves addressing several key difficulties related to controlling hardware behavior, maintaining measurement accuracy, and dealing with various system-level constraints. By carefully designing the benchmarks and their supporting infrastructure, it is possible to overcome these challenges and ensure the reliability and relevance of the collected performance data.

Balancing Precision and Portability

Achieving precision while preserving portability is a central concern in microbenchmark design. Although assembly code provides direct control over the processor's operations and can precisely induce specific events, it is closely tied to a particular architecture. Adapting a microbenchmark to different platforms requires effort due to variations in instruction sets and hardware configurations. To mitigate this issue, the microbenchmarks are structured as modular, parameterized code blocks. By keeping architecture-dependent code isolated, the adaptation process simplifies to redefining only the platform-specific sections. Clear documentation and guidelines further streamline this process, ensuring that developers can apply the same measurement methodologies across a range of systems.

Ensuring Repeatability in a Noisy Environment

In real-time multicore systems, external factors such as background tasks, interrupts, and fluctuating resource availability can introduce noise that affects the repeatability of results. To reduce these effects, the microbenchmarks are executed under controlled conditions. On bare-metal setups, running the benchmarks in isolation prevents interference from concurrent tasks. In RTOS-based environments, careful configuration of priorities and scheduling policies helps limit external disturbances. Repeated runs of each benchmark allow for statistical analysis that identifies and filters out anomalies, ensuring that the resulting performance measurements more accurately reflect the intended conditions.

Accurate Event Generation and Validation

A key objective of microbenchmark design is to reliably produce the targeted hardware events—such as cache misses or bus accesses—without unwanted side effects. Accomplishing this requires a thorough understanding of the underlying architecture to select addresses, instructions, or operations that provoke the desired response. Once implemented, the benchmark's behavior is validated by comparing observed event counts with expected outcomes. This validation step confirms that the benchmark is correctly inducing the events of interest and that the profiling framework accurately captures them, lending credibility to subsequent analyses.

Minimizing Intrusiveness of Instrumentation

Instrumentation must be applied carefully to avoid altering the execution profile in ways that skew results. Software instrumentation relies on lightweight instructions inserted at carefully chosen points, minimizing additional overhead. Hardware-based methods, such as using the DSU's debugging capabilities, capture traces with minimal interference. After data collection, post-processing filters out irrelevant or redundant information, preserving only the details that contribute to meaningful performance insights. By controlling instrumentation overhead and data quality, it becomes possible to trust the integrity of the final measurements.

Handling Resource Constraints

Embedded and real-time environments often impose strict limitations on memory, processing power, and other system resources. Crafting microbenchmarks that fit within these constraints without sacrificing event generation quality demands careful optimization. Techniques such as loop unrolling, data alignment, and judicious control of memory usage help maintain a high density of targeted operations while respecting the system's boundaries. By balancing workload intensity with resource availability, the microbenchmarks remain both effective and compatible with the hardware under scrutiny.

Reconciling Timing Analysis with Hardware Limitations

On platforms without advanced monitoring features, such as a PMU, obtaining precise profiling data becomes more complex. Here, the debugging capabilities of components like the DSU are employed to gather instruction or bus-level traces. Although this approach may be more time-consuming and less direct, post-processing these traces can still yield accurate performance metrics. The result is a flexible framework that demonstrates that comprehensive timing analysis and profiling remain feasible, even on hardware with limited introspective capabilities.

4.4 Evaluation of Microbenchmarks Precision

After designing and implementing the microbenchmarks, it is essential to assess their precision in triggering the intended hardware events and minimizing unintended side effects. The ultimate goal is for each microbenchmark to produce at least 95% of the target event occurrences (allowing for a 5% margin of deviation, which is ultimately unavoidable by construction) while suppressing or reducing other events that could dilute or confound the measurements. For example, a microbenchmark designed to induce L2 store misses should not inadvertently generate a significant number of L2 load misses, as that could generate contention but not due to the intended hardware event. Maintaining this focus ensures that the data collected from the microbenchmarks is both meaningful and actionable for subsequent profiling and timing analyses.

To achieve this precision, the design process involves extensive experimentation and tuning. Parameters such as memory access patterns, data structure sizing and alignment, and loop iteration counts are carefully adjusted to consistently yield the desired event. The microbenchmarks are repeatedly exercised under various conditions to confirm the stability of their behavior, eliminating any potential jitter effects. By iterating through these refinements, the benchmarks evolve toward stable, high-quality tools that produce the intended events with minimal noise.

While the conceptual principles and design considerations for achieving such precision are discussed here, the concrete evaluation results and exact performance figures are presented later in the thesis. In particular, detailed validation experiments, including their numerical results in a real use case example, are provided in Chapter 8. There, Table 8.3 and related experiments show how closely the observed outcomes align with the expected values for specific microbenchmarks in a practical setting. These experiments involve carefully selected code snippets and memory access patterns that are straightforward for a hardware expert to analyze and predict, thus serving as a reliable reference for validating the profiling methodology.

By comparing the expected and observed event counts, the evaluation in Chapter 8 demonstrates that the microbenchmarks can indeed achieve the desired precision targets. The data reveals scenarios where the deviation is small enough—well within the 5% target margin—to validate both the microbenchmarks themselves and the overarching profiling

framework. The results from those experiments confirm that, for benchmarks targeting particular events like cache hits, cache misses, or memory bus accesses, the induced patterns closely match the intended behaviors, reinforcing the soundness and utility of the microbenchmarking strategy.

4.5 Slowdown Matrix Concept

As discussed throughout previous chapters, in multicore systems, multiple cores frequently compete for shared resources such as caches, memory buses, or off-chip memory interfaces. This contention can significantly impact the latency of certain operations. For example, an L2 load miss that completes in 10 cycles when run in isolation may require 20 or more cycles if another core is simultaneously issuing similar or conflicting operations. To systematically capture and quantify these contention effects, we introduce the concept of a *slowdown matrix*.

A slowdown matrix encapsulates the additional latency incurred by a particular type of event—such as an L2 load miss—when one or more other events occur simultaneously on separate cores. Each entry in the matrix represents a scenario where a given event (e.g., an L2 read to off-chip SRAM) is executed concurrently with a specific type of contender event (e.g., another core performing heavy writes to the same memory). By comparing these conditions against the baseline, single-core scenario, the matrix expresses how each combination of events influences overall execution time.

The construction of the slowdown matrix involves several key steps. First, we measure the cost of each event type in isolation, ensuring that no other cores introduce interfering operations. This baseline establishes the reference latency for each event—such as the 10 cycles needed for an L2 load miss in a quiescent system. Next, we design and run stressing benchmarks on additional cores to induce maximum contention for the shared resource in question. These benchmarks, deliberately engineered to saturate the resource, are executed concurrently with the target event, allowing us to observe and record the resulting latency increase. Repeating this procedure for different pairs or sets of events populates each cell of the slowdown matrix with the measured latency under contention.

The reliability and usefulness of the slowdown matrix hinge on the quality of the stressing benchmarks and the accuracy of the event measurements. The more effectively a benchmark stresses the resource, the more representative the resulting slowdown values will be. Consequently, producing a robust slowdown matrix depends on both thorough event characterization (as discussed in Section 8.4) and careful benchmark design.

While this section provides the conceptual framework for understanding slowdown matrices, a concrete example and detailed evaluation results are presented later in Chapter 8.5. There, you will find a fully constructed slowdown matrix (Figure 8.3) for the GR712RC platform. This matrix highlights how different memory accesses—ranging from on-chip SRAM reads to off-chip SDRAM writes—experience varying degrees of slowdown when contending events occur on other cores. These examples illustrate how slowdown matrices can be used to identify worst-case latencies, guide system-level optimizations, and improve

the predictability and reliability of multicore systems in real-world scenarios.

4.6 Stress Testing the Platform and Applications

Once microbenchmarks have been defined, validated, and used to generate the slowdown matrix, their utility extends beyond mere characterization of resource contention. The slowdown matrix offers valuable insights into how much an application can be delayed under the worst possible contention scenarios. Armed with this understanding, system engineers can employ microbenchmarks to stress-test both the platform and specific applications running on it, effectively simulating conditions that approximate or even exceed worst-case operational environments.

The fundamental idea behind this approach is to complement theoretical timing analysis with empirical evidence. Timing analysis alone—especially in multicore systems where shared resources and complex contention patterns are prevalent—can be overly conservative if it simply assumes the worst possible interference without concrete data. The slowdown matrix provides the missing link, offering a quantifiable way to reason about how different events interact and how these interactions translate into extended execution times.

For instance, consider an application that is sensitive to off-chip write contention. From the slowdown matrix, one might deduce that concurrent off-chip writes from another core increase the latency of the application's critical operations by a factor that could threaten its ability to meet a hard real-time deadline. The next logical step is to put theory into practice. By deploying a dedicated stressing microbenchmark on a spare core—one engineered to produce a high density of off-chip writes—the platform can be pushed into a scenario that closely mirrors the identified worst-case conditions. Running the target application concurrently with this stressing benchmark offers a practical evaluation of how the application's timing characteristics change. If the application still meets its deadlines under these contrived but carefully controlled test conditions, engineers gain confidence that the theoretical worst-case scenario, as predicted by timing analysis, is indeed safe.

This methodology is critical for developing dependable real-time systems. Modern multicore platforms exhibit intricate resource-sharing behaviors that are not easily captured by static analysis alone. The synergy between microbenchmark-driven stress tests and the slowdown matrix enables a far more tangible validation process. By inducing carefully curated patterns of contention, it becomes possible to confirm (or refute) the worst-case bounds predicted by theoretical models. If discrepancies arise, these results guide refinement of the models or highlight the need for additional mitigation techniques. In practice, this capability allows system developers to:

1. **Validate Timing Analysis Assumptions:** Confirm that the WCET estimates, derived from theoretical or analytical models, hold true in actual hardware conditions.

2. **Identify Potential Vulnerabilities:** Detect scenarios where real-world contention may surpass the predictions, highlighting areas where further optimization, buffering, or resource partitioning may be required.

3. **Optimize for Safety Margins:** Fine-tune the system's configuration and scheduling policies to ensure that even under extreme contention, all tasks continue to meet their deadlines.

Ultimately, without the ability to recreate and measure these adverse conditions, designers would have to rely solely on guesswork or conservative assumptions. Microbenchmarks and the slowdown matrix bridge this gap, providing the means to translate abstract architectural insights into concrete, testable scenarios. This approach closes the loop from theoretical timing analysis to empirical validation, ensuring that when a system is certified as "safe" under worst-case conditions, that safety claim is backed by robust, data-driven testing and verification.

Chapter 5

Design and Solution

This chapter is dedicated to the design and solution strategies developed to address the challenges of multicore contention in real-time embedded systems. We start by introducing the concept of time composability, a critical property that ensures predictable system behavior when components are integrated. Understanding this concept is fundamental as it underpins the methodologies proposed in this thesis.

Next, we delve deeper into the access pairing technique, which serves as the backbone of the two proposed methods. This technique is pivotal in accurately modeling and managing contention effects by pairing access types and analyzing their interactions.

Following this, we present a reference architecture that aligns with the system model described in previous chapters. This reference architecture serves as a concrete platform for implementing and validating our proposed methods, providing a realistic context for our experiments and evaluations.

Finally, we introduce the rationale behind the two proposed methods: an iterative approach and an ILP-based approach. We discuss how these methods, while different in their mechanisms and applications, are complementary. Each method offers unique advantages and together, they provide a comprehensive solution to the problem of multicore contention in critical real-time systems. The iterative approach offers flexibility and adaptability, while the ILP-based approach provides precise and optimal solutions. This duality ensures that our proposed strategies can be effectively applied across a broad range of scenarios and system configurations.

5.1 Time Composability in Multicore Real-Time Systems

Time composability is a fundamental property in the domain of real-time and multicore systems. It paves the way for modular development and verification processes by ensuring that the timing behavior of individual software components remains consistent and predictable, irrespective of the behaviors of other components in the system. Let's delve deeper into this concept and its significance. At its core, time composability refers to the ability

to analyze and validate the timing behavior of a software component in isolation and then compose these individual analyses to predict the entire system's timing behavior. In other words, the timing analysis of a component, when executed alone, should match its behavior when executed alongside other components. When applied to a multicore processor, this can be translated into the property that the timing behavior of a task (or at least an upper bound thereof) is not affected by the activity of its co-runners [34]. However, due to the inter-core dependencies, we must account for highly variable-timing behaviour, hence we can not straightforwardly assume that safe timing bounds in isolation will also hold in a multicore architecture. Therefore, it must be guaranteed that the assumptions on timing behaviour are still valid under the composition with a new set of tasks.
In real-time systems, where tasks have strict timing deadlines, the predictability of execution times is paramount. Time composability aids in:

- **Modularity:** Developers can design, analyze, and verify components independently, streamlining the development process.
- **Reusability:** Components verified in one system can be reused in another, without the need for re-analysis.
- **Scalability:** As systems grow in complexity, being able to add components without re-analyzing the entire system becomes crucial.

While time composability is a beneficial property, achieving it in multicore environments poses unique challenges. Shared resources, such as caches, buses, and memory interconnects, become contention points. When tasks from different cores attempt to access these resources simultaneously, it can lead to unpredictable timing delays, compromising the desired compositional properties. Proper management strategies and architectural designs are necessary to mitigate these challenges and ensure that multicore systems maintain time composability.

In the realm of real-time and embedded systems, the concepts of Full Time Composability (fTC) and Partial Time Composability (pTC) play crucial roles in ensuring the predictable and reliable performance of complex computing systems. These two approaches address the challenges associated with designing systems where multiple critical tasks, each with unique timing requirements, need to coexist and execute harmoniously.

- **Full Time Composability (fTC)**: It represents the ideal scenario where the timing behavior of each component is always consistent and predictable, irrespective of the presence or behavior of other components in the system. Systems that achieve fTC: (1) Ensure that components' timing behaviors remain unchanged regardless of system-wide interactions, (2) Provide strong guarantees about worst-case execution times and interactions with shared resources and finally (3) Feature stringent isolation mechanisms, which can come at the expense of decreased resource efficiency or

the introduction of additional system overhead. While fTC is the gold standard for time composability, achieving it, especially in multicore systems, requires meticulous design, architecture, and validation strategies.

- **Partial Time Composability (pTC)**: It is a relaxed version of the time composability principle, tailored to scenarios where full time composability might be too restrictive or challenging to achieve. In systems exhibiting pTC: (1) Components' timing behavior is predictable and composable under a subset of system scenarios or conditions, (2) Predictable interactions among components are allowed, but only to a certain extent or within certain bounds, and finally, (3) pTC can facilitate improved resource utilization and system performance compared to striving for full time composability, especially in complex multicore systems.

5.2 WCD Bounding Based on Access Pairing

In order to conservatively but accurately capture the effect of different types of accesses, the concept of *access pairing* will be used. This refers to how we align task overlapping and collide tasks' bus accesses in order to derive contention. Pairing models the fact that requests from different cores can collide in the access to a shared hardware resource. Contention causes an increase in the execution time of the interfered task: the impact on the WCET of the latter is bounded by pairing victim and culprit accesses. Accesses of interfering tasks will cause a contention delay (up to the interfering access latency) on the interfered task for each access they can be paired with. Note that pairing can only happen when the interfered task and the interfering task, in different cores, overlap in time (i.e. run in parallel). As an example, task τ_1 in Core 0, in the topmost part of Figure 5.1(a), performs two memory accesses, symbolized with $\bullet$, that can be paired with two accesses in τ_3, executing in parallel in Core 1. Once paired, these two accesses in τ_3, will not be available for pairing with other accesses from Core 0. In consideration of the fact that contenders' accesses may exhibit different latencies depending on the access type, it is important to conservatively pair accesses starting from the most interfering ones (higher latency).

Access pairing, as a metric to compute contention, and tasks overlapping are in a *mutual relationship* as pairing is determined on the accesses from the set of overlapping tasks, and the latter is affected and potentially altered by the increase in execution time caused by the WCD. At a system level, considering all tasks in each core, such relationship can be sometimes counterintuitive as local worst-case pairing for a task may lead to a shorter makespan. Figure 5.1 provides an illustrative example of such effect, avoiding any consideration on access types, for the sake of simplicity. Tasks τ_1, τ_2 execute on Core 0 and perform respectively 2 and 5 memory accesses, while tasks τ_3, τ_4, τ_5 execute on Core 1, with 4, 1 and 1 memory accesses, respectively (see Figure 5.1(a)). In this example, let's focus on the WCD of tasks running on Core 0 and the produced makespan. In principle, 6 out of 7 accesses in τ_1 and τ_2 can potentially be paired with accesses from tasks running on

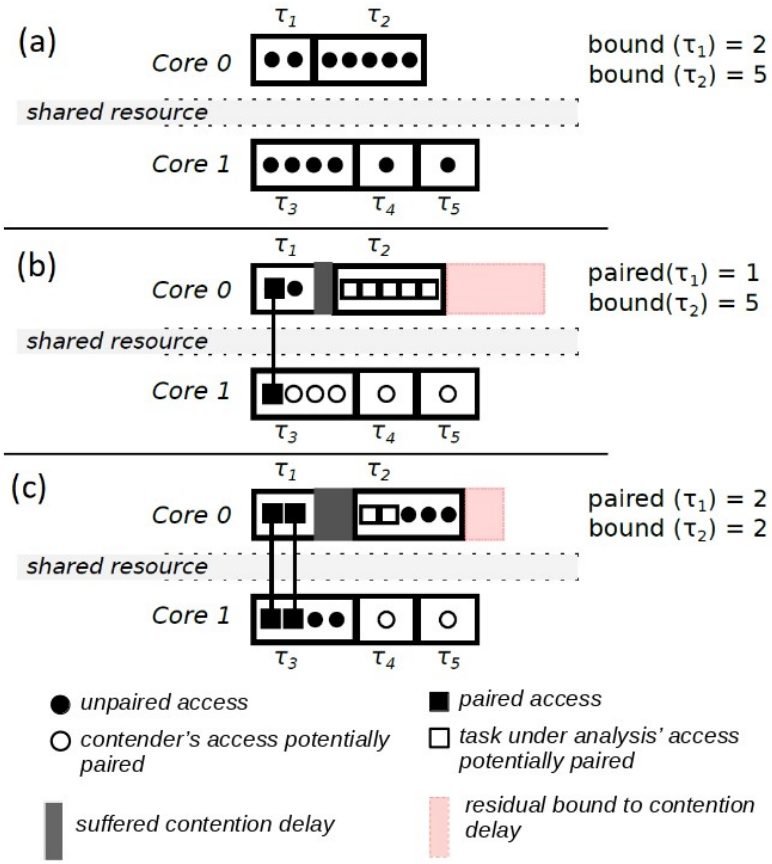


Figure 5.1: Access pairing: motivation

Core 1. However, this ultimately depends on how tasks overlap in time, which is influenced by how much contention tasks suffer. Let's assume a locally-good scenario, Figure 5.1(b), in which only 1 access in τ_1 is paired with an access in τ_3 (i.e. τ_1 execution time will suffer a delay while waiting for τ_3 to perform its 'paired' access). The resulting overlapping is still compatible with τ_2 having all its 5 accesses paired with accesses from τ_{3-5}. The local worst-case scenario, where τ_1 has all its accesses paired, instead, leads to a task overlapping that is compatible with τ_2 having at most 2 accesses paired with accesses from τ_{4-5}. Thus a local worst-case assumption leads to a shorter, potentially optimistic worst-case makespan.

As a motivation to this work, this simple example shows that trying to compute the WCD pursuing the local worst-case conflicts, on a purely per-task basis, can lead to unsafe system-level bounds. In this thesis we present two safe approaches for the computation of the WCD:

- As part of the iterative proposed method, we avoid the above limitation by not focusing exclusively on the local worst-case. When considering a pair of overlapping tasks, we still capture the local worst-case pairing of accesses, but we do not prevent

already paired accesses to be paired with accesses from other tasks on the same core. Considering the example in Figure 5.1, we allow τ_3 accesses to be paired with τ_2 ones, despite 2 τ_3 accesses were already paired with τ_1. This is indeed an unrealistic condition as we know that one access cannot conflict with more than one access triggered on the one and same core; however, it conservatively guarantee safe WCD results. The resulting approach proceeds iteratively, propagating the effect of WCD computation on task overlappings, and determining the worst access pairing scenarios under given task alignments. In order to determine the initial task alignment as well as the ones after each iteration, different assumptions will be described later in this chapter.

- The second method builds on the observation that tasks overlapping conditions and access pairing lend themselves to be captured with an Integer Linear Programming (ILP) formulation, to compute the worst (longest) possible makespan. Hence, an ILP formulation is used to model and explore all possible (feasible) combinations of tasks overlapping and access pairing, thus providing a safe system-level bound (considering pairing across task boundaries) to the worst-case contention delay. System level bounds are particularly relevant in those real-time systems whose execution is clearly organized as the cyclic succession of time frames, such as cyclic-executive and Integrated Modular Avionics (IMA) systems, which are the main focus of this work.

To compute the WCD, we will make use some notation to identify the different pairing events. At any moment, τ_j is assumed to generate interference on τ_i: i.e., τ_j's request always precedes τ_i.

In this section we focus on the definition of accesses and pairing thereof. For what concerns task definition and notation, we assume the reader to be familiar with the notation introduced in Table 3.1. The WCD caused by τ_j on τ_i is computed based on the number (and type) of paired accesses since the latter are conservatively assumed to always generate interference on τ_i. To compute the WCD for τ_i we need to consider all possible access parings for τ_i's accesses, which depend on, and at the same time potentially affect, the set of (overlapping) contender tasks τ_j. The WCD analysis is performed per contender core (i.e., $\mathcal{T}_s : \pi_s \not\uparrow\downarrow \tau_i$) and per MIF. The interference suffered due to each core are summed up to derive the global WCD for τ_i. When not otherwise specified, this analysis applies at the granularity of scheduling intervals or MIFs: the analysis has to be separately applied to all MIFs in the MAF. Since the model and its formulations apply to a given MIF, overloading the notation is avoided and an indicator of the considered MIF is not added either.

Table 5.1 illustrates the notation employed to describe the items that will be used to define the effect of contention.

The tightness of both approaches directly depends on their capability to exclude infeasible pairings. For that reason, $\bowtie_i(\mathcal{T}_s)$ is defined, as not every single task of the system can be a contender to a certain task under analysis (due to running at different instants under any possible contention scenario).

Worst-case delay computation	
$\bowtie_i(\mathcal{T}_s)$	Set of tasks $\mathcal{T}_s$ mapped to core $\pi_s \not\leftrightarrows \tau_i$, potentially overlapping with the task under analysis τ_i
$a^t_{j\triangleright i}$	Number of accesses of type t in $\tau_j \in \bowtie_i(\mathcal{T}_s)$ that are *paired* with accesses in τ_i
$a_{j\triangleright i}$	Total number of accesses (of any type) in $\tau_j \in \bowtie_i(\mathcal{T}_s)$ that are *paired* with accesses from τ_i.
$\Delta_{j\triangleright i}$	Interference suffered from τ_i because of contention triggered by *paired* accesses of **any** type in τ_j
Δ_i	Overall interference suffered from τ_i
$c_i + \Delta_i = e_i$	Execution-time budget reserved for multicore execution of τ_i after accounting for WCD effects

Table 5.1: WCD computation notation used throughout the thesis

Furthermore, as it can be noted, $\bowtie_i(\mathcal{T}_s)$ is a reflexive relation: if task τ_i runs in parallel with task τ_j, naturally τ_j does so with τ_i: $\tau_j \in \bowtie_i(\mathcal{T}_s) \iff \tau_i \in \bowtie_j(\mathcal{T}_{s'})$. Further, since tasks running on the same core do not interfere each other, it holds that $\tau_i \in \mathcal{T}_s \Rightarrow \bowtie_i(\mathcal{T}_s) = \emptyset$.

When considering access pairings, slightly different assumptions may result in different solutions, even within the same approach, as we shall see with some examples. While some constraints are particular to a specific approach, some constraints are shared between the iterative and ILP approaches. Next we identify those constraints on access pairings that apply to both approaches.

Bounds on task-to-task access pairing

The total number of accesses in τ_j *paired* (hence conflicting) with accesses in τ_i is bounded by the access counts in both tasks:

$$a_{j\triangleright i} \leq \min(a_i, a_j) \tag{5.1}$$

where $a_{j\triangleright i}$ is a generalization of $a^t_{j\triangleright i}$ and indicates the total accesses (of any type) in $\tau_j \in \bowtie_i(\mathcal{T}_s)$ that are *paired* with accesses from τ_i. When considering different access types, the above constraint can be redefined on a per-type basis (to allow a consistent pairing of latencies) as follows:

$$\sum_{t\in\Gamma} a^t_{j\triangleright i} \leq \min(a_i, a_j) \tag{5.2}$$

More precisely, the above equation can be further constrained by considering that the

number of paired accesses of a given type in the interfering task (τ_j) is limited by the number of accesses of that type in the interfering task and the number of accesses in the interfered task:

$$a^t_{j \triangleright i} \leq \min(a_i, a^t_j) \tag{5.3}$$

Bearing in mind the constraint defined by Equation 5.3, let's present the two approaches in more detail.

In Sections 6 and 7, we will further elaborate on how these assumptions are factored in the proposed approaches for the computation of the worst-case contention delay.

5.3 Architecture: LEON4

Both the iterative and the ILP-based approaches are parametric with respect to admitted access types and respective maximal latencies. In order to run our experiments, we considered the access types and associated latencies exhibited by a concrete hardware platform, while being compatible with the hardware assumptions we made in Chapter 3. The Cobham-Gaisler 4-core LEON4 platform [3], extensively used in the embedded space domain, has been selected as the reference platform to conduct our experiments (both for feeding the models and for running the proof of concept evaluation). In particular, we focused on an FPGA implementation of the LEON4, and exploited the available performance monitoring counters (PMCs) to extract useful information from the tasks memory accesses profiles, necessary to the WCD computation models. The main reason for selecting this platform, is that it is highly representative for the space domain, as it has been extensively adopted in several European Space Agency activities.

The LEON4 (see Figure 5.2) comprises 4 homogeneous cores operating at 250MHz, all counting on L1 private caches for data and instructions. An AMBA bus, implementing round-robin arbitration, connects the four cores to a shared L2 cache, which can be configured to support different associativity levels (1/2/4), way size (1-256 KB) and AHB bus width (64 to 128 data bits). In the considered configuration, the L2 cache is partitioned, which means that each core counts on its own L2 cache partition to avoid further, inordinate timing interference among tasks in different cores. The L1 data cache implements a write-through, no-write-allocate policy, while the L2 cache is write-back. Requests in the bus block the execution until they are completely served. For that reason, the bus is the main source of contention in this architecture.

5.4 Deriving Requests & Latencies in the LEON4

In the presented LEON4 architecture, different types of memory request can go through the bus to reach the L2 (and possibly the main memory). The following access types are possible in the platform [26]: L2 store hits ($s2h$) and load ($l2h$) hits, L2 clean and dirty

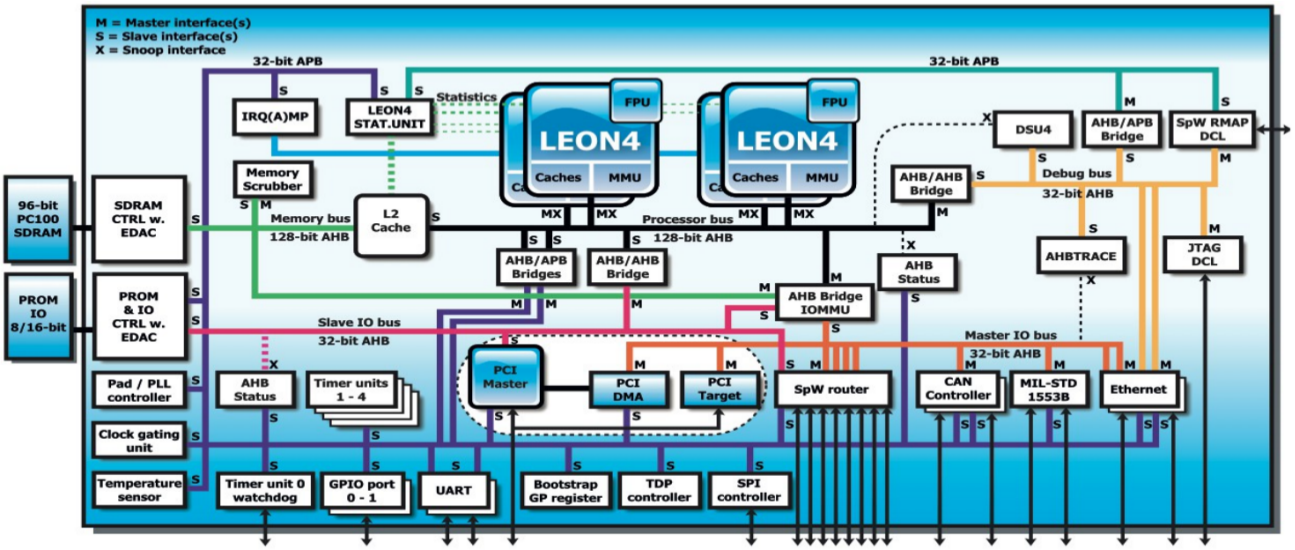


Figure 5.2: LEON4 development board [4]

misses triggered by load (*l2mc*, *l2md*) and store (*s2mc*, *s2md*) operations. Access types were directly (or analytically) monitored by the performance monitoring counters, while latencies per access type were already experimentally derived in a recent work by Diaz *et al.* [26], exploiting the available PMC support. Table 5.2 summarizes the maximum latencies experimented when each of the described events happen.

Type	mcd [cycles]	Description
sh	$l^{sh} = 1$	L2 store hit
lh	$l^{lh} = 8$	L2 load hit in L2
lmc	$l^{lmc} = 28$	L2 load clean miss
smc	$l^{smc} = 28$	L2 store clean miss
lmd	$l^{lmd} = 31$	L2 load dirty miss
smd	$l^{smd} = 31$	L2 store dirty miss

Table 5.2: Latencies of request type

Since l^{lmd} and l^{smd} bring up the same maximum contention delay, this contention will be indifferently referred to as $l^{\max}$.

In order to monitor the required events, four PMCs from the LEON4 debug support unit have been used, which means that for sure not all relevant events can be directly monitored. Concretely, Table 5.3 lists the bus access types that can be captured with the available PMC support. As can be observed by comparing Tables 5.2 and 5.3, not all access types can be directly tracked with the current PMC support. However, missing counters can be analytically (and conservatively) derived.

PMC	Description
pmc^{icm}	Bus reads caused by instruction cache (ic) misses
pmc^{dcm}	Bus reads caused by data cache (dc) misses
pmc^{st}	Writes to L2
pmc^{m}	Misses in the L2

Table 5.3: PMCs available in the reference processor

Table 5.4 below shows all the possible events that are related to bus accesses. Note that *"l"* and *"s"* stand for load and stores respectively in the superindexes, *"h"* and *"m"* do for hit and miss, and *"c"* and *"d"* stand for clean and miss. The maximum contention delay (mcd) is measured in clock cycles. As already observed, with the current PMC support, we are not able to track all the above events. However, we show how we can analytically derive (or upper-bound) the untraceable events from the available PMCs with a simple set of formulas.

L2 cache	hits (n^h)	misses (n^m)	
		dirty (n^{md})	clean (n^{mc})
loads (n^l)	n^{lh}	n^{lmd}	n^{lmc}
stores (n^{st})	n^{sh}	n^{smd}	n^{smc}

Table 5.4: L2 cache events

First, we observe that the events monitored by the L2 miss counter (pmc^m) do belong to four types of misses, depending on the operation type. Equation 5.4 unfolds the total number of L2 cache misses depending on whether or not the cache miss was triggered in response to a load or store operation, and whether or not it was involving the write back of dirty cache content (dirty or clean miss).

$$pmc^m = n^{lmc} + n^{smc} + n^{lmd} + n^{smd} \tag{5.4}$$

With the current support we are not able to derive the precise number of dirty misses that occurred. Since dirty misses imply a larger delay than clean misses, it is important to be able to precisely (and safely) upper bound their number. In order to do so, we exploit the fact that a dirty miss can only happen if it has been preceded by a store operation (and there cannot be more dirty misses than misses). Equation 5.5 safely upper bounds the number of dirty misses by considering the minimum between store operations (available as pmc^{st}) and the overall number of L2 misses (pmc^m).

$$\hat{n}^{md} = min(pmc^m, pmc^{st}) \tag{5.5}$$

Consequently, Equation 5.6 uses the bound on dirty misses to derive a lower bound on the number of clean misses (n^{mc}).

$$\check{n}^{mc} = pmc^{m} - \hat{n}^{md} \tag{5.6}$$

In the considered platform, the latency incurred by a cache miss does not depend on the operation type (i.e., load or store). The dirty miss latency is identical for both loads and stores (31 cycles), and the same goes for clean misses (28 cycles). Therefore, it is not necessary to differentiate them.

Instead, since L2 load hits produce a worst case delay of 8 cycles whereas store hits of only 1, the number of load hits must be safely upper bounded. By applying the same reasoning we followed for bounding the number of misses, we observe that there cannot be more load hits than the number of hits or loads themselves. As a result, Equation 5.7 upper-bounds the number of load hits by considering the minimum between the overall number of L2 hits (n^h) and the number of loads (n^l). Note that the former can be derived by subtracting the number of L2 misses from the total number of bus accesses, while the latter can be derived by summing L1 instruction and data cache miss counters.

$$\begin{aligned} n^h &= pmc^{icm} + pmc^{dcm} + pmc^{st} - pmc^{m} \\ n^l &= pmc^{icm} + pmc^{dcm} \\ \hat{n}^{lh} &= min(n^h, n^l) \end{aligned} \tag{5.7}$$

The remaining store hits are thus modeled by Equation 5.8 below:

$$\check{n}^{sh} = n^h - \hat{n}^{lh} \tag{5.8}$$

Exploiting the above formulas we can derive all four relevant events that influence the latency of requests in the bus, which is exactly the information we need to evaluate the presented approaches.

In this Thesis we present two safe approaches for the computation of the WCD:

- As part of the iterative proposed method, we avoid the above limitation by not focusing exclusively on the local worst-case. When considering a pair of overlapping tasks, we still capture the local worst-case pairing of accesses, but we do not prevent already paired accesses to be paired with accesses from other tasks on the same core. Considering the example in Figure 5.1, we allow τ_3 accesses to be paired with τ_2 ones, despite 2 τ_3 accesses were already paired with τ_1. This is indeed an unrealistic condition as we know that one access cannot conflict with more than one access triggered on the one and same core; however, it conservatively guarantee safe WCD

results. The resulting approach proceeds iteratively, propagating the effect of WCD computation on task overlappings, and determining the worst access pairing scenarios under given task alignments. In order to determine the initial task alignment as well as the ones after each iteration, different assumptions will be described later in this chapter.

- The second method builds on the observation that tasks overlapping conditions and access pairing lend themselves to be captured with an Integer Linear Programming (ILP) formulation, to compute the worst (longest) possible makespan. Hence, an ILP formulation is used to model and explore all possible (feasible) combinations of tasks overlapping and access pairing, thus providing a safe system-level bound (considering pairing across task boundaries) to the worst-case contention delay. System level bounds are particularly relevant in those real-time systems whose execution is clearly organized as the cyclic succession of time frames, such as cyclic-executive and Integrated Modular Avionics (IMA) systems, which are the main focus of this work.

Chapter 6

Iterative Approach

In this section, the iterative approach is presented. First, we will go through its foundations and principles. Then, we will describe in detail the steps to be followed for its application. We will enumerate the required assumptions and provide a simple and short example of the application process. Later, we will prove the safeness of the approach and finally show some experimental results.

6.1 Task-Level Guarantees: The Iterative Approach

This subsection formally describes the iterative approach in detail. We will start by discussing its foundational principles and then describe the step-by-step application process. This includes outlining the required assumptions and providing a concise example of the method in action. Finally, we will demonstrate the safety and reliability of this approach.

The iterative approach focuses on computing reliable Worst-Case Delay (WCD) bounds at the task level. This means it calculates the maximum contention that each task can experience under certain assumptions. By aiming for task-level guarantees rather than system-level ones, we accept the trade-off of potentially larger makespans. Unlike other methods, an access from a contending task is not immediately discarded when it overlaps with another task; instead, it is considered across all overlapping tasks. This approach is necessary due to the uncertainty of which specific task will be contending at any given time. Additionally, when accesses from different tasks might occur simultaneously, each one is assumed to be the last to gain access to the shared resource. This conservative assumption is made because it is not possible to guarantee the order of access service at runtime, thus requiring us to consider all possible scenarios.

These conservative assumptions introduce some level of pessimism at the makespan level. Specifically, we identify two main sources of overestimation in the iterative approach. (i) **Access Alignment**: The approach assumes that if an access can cause contention, it will, which may not always be true due to runtime variations. And (ii) **Over-Accounting**: When a contender's access overlaps with multiple tasks in the analyzed core, the iterative approach assumes it contends with all of them, which can lead to overestimation.

Despite these conservative assumptions, the iterative approach aims to calculate safe triggering times for each task, ensuring that the previous job on that core has completed, even under maximum contention. By setting appropriate triggering times, we reduce the number of feasible overlapping scenarios, thus maintaining a moderate computational complexity.

Algorithm 2 formally presents the iterative approach. The types of bus requests and associated latencies are read from a configuration file, making the approach modular and adaptable to various architectures and events.

Initially, tasks on each core are characterized, and the initial task alignment is identified. Without any contention, τ_i's budget is c_i. The iterative process begins by checking the alignment of co-running tasks and pairing their accesses. In the first iteration, r_i occurs immediately after c_{i-1} for each task. The contention caused by this alignment is calculated iteratively, incrementally pairing each task's access with the most contending one available. This process continues for all tasks, updating budgets and release times, potentially leading to a new alignment scenario. The process repeats until a fixed point is reached, where further iterations do not change the pairings and budget inflations.

Algorithm 2 ITERative Approach for WCET Computation

```
1: procedure CalculateWCET(cores, tasks, events, latencies)
2:     Read task execution profiles, bus access patterns and their latencies
3:     for each core in cores do
4:         Characterize bus requests for each task
5:     end for
6:     Identify initial tasks alignment                ▷ Initially, r_i based on c_{i-1}
7:     Initialize the iterative process
8:     repeat
9:         for each core in cores do
10:            for each task in core do
11:                Calculate contention based on task pairings
12:                Inflate task under analysis' budget
13:                Update next task's release time based on task under analysis' contention
14:            end for
15:        end for
16:        Check for a fixed point where no contention changes occur
17:        if fixed point not reached then
18:            Update task execution times with new contention data
19:            Identify new task pairings considering updated times
20:        end if
21:    until fixed point reached
22:    return WCET for each task computed
23: end procedure
```

In each iteration, the execution budget of every task may be inflated to account for the worst-case contention with co-running tasks. This conservative strategy arises from the unpredictability inherent in real-time multicore systems, where it is challenging to precisely predict which tasks will contend at runtime. By iteratively expanding the execution budgets to reflect potential mutual contention across all cores, the algorithm converges to a state where no further budget increases are necessary. This point of convergence occurs when the algorithm's conservative assumptions have sufficiently accounted for all potential contention scenarios, ensuring that the calculated execution budgets are robust against any actual contention that may arise. The algorithm's convergence is guaranteed, as the process is bounded by the finite number of tasks and contention possibilities, each of which is considered during the iterative budgeting.

6.2 Time-Composable Worst-Case Delay Bounds

Time composability, as introduced in Section 5.1 refers to the ability to analyze and validate the timing behavior of individual software components in isolation and then compose these analyses to predict the entire system's timing behavior. In multicore systems, this implies that the timing behavior of a task (or its upper bound) is unaffected by the activities of its co-runners [34]. However, due to inter-core dependencies, timing behavior can vary significantly, necessitating guarantees that these assumptions remain valid when new tasks are added.

A strict way to meet this requirement is through **full composability**, which ensures that system requirements, such as timing bounds, hold regardless of the other components in the system. For multicore timing analysis, this means that all task deadlines will be met irrespective of the contending tasks and without further restrictions or assumptions. Achieving fully time-composable contention bounds requires assuming the maximum possible contention at each instance: every time tasks access the bus, the maximum number of contenders with the highest latencies is encountered, and the task under analysis is the last to gain access, with each contender using the shared resource for the maximum possible time.

The concept of full time composability (fTC) is explored in [34] and further studied in [26], under the term *fully Time-Composable* (fTC) bounds. Although fTC is safe by design, it is highly pessimistic, as will be demonstrated in Chapter 7.5. This pessimism becomes evident when fTC is used as a baseline for comparison with the iterative approach presented in this Thesis.

Time composability is often viewed as an all-or-nothing metric [34]: a system either meets its timing guarantees with a set of components or it does not. To avoid overly pessimistic and unusable WCD bounds, a **partially** time-composable guarantee can be provided by limiting the spectrum of possible contenders and ensuring that the provided bounds are valid for this specific set of components. This approach guarantees deadlines for a certain set of co-running tasks, even under the worst possible alignment, but it requires information about the actual contenders at runtime. The authors in [26] offer

an alternative formulation to derive more realistic bounds without compromising safety guarantees, calling this approach *partial Time Composability* (pTC). pTC considers only the number and type of accesses to the bus performed by the actual set of potentially overlapping tasks.

The pTC paradigm underlies our approaches, with the iterative approach presented in this thesis computing pTC bounds. This will be further described in the remainder of this chapter. An important factor in pTC is that the hardware resource (in this case, the bus) supports different access types. Ignoring this would require assuming that each access always incurs the worst-case latency, which is not generally true. We explicitly recognize that task requests can exhibit variable latencies depending on the request type and take these into account to conservatively but tightly compute the worst-case contention effects.

6.2.1 fTC Contention Model

To derive an fTC WCD bound, we have to assume that every single access to a shared resource that a particular task performs will encounter the maximum number of contending tasks. That is, $|\Pi|-1$, since only one task per core can be contending at a precise instant, and the contention delay will be maximum (mcd) as well. This guarantees that no runtime scenario will be able to cause a worse delay. The fTC model can serve as an upper bound to the contention and slowdown that a task can suffer.

Therefore, to compute the WCET of task τ_i (e_i) under the fTC assumption, we will have to assume that every single access to the bus (a_i) incurs the maximum penalty possible in the system ($l^{\max}$). This is formalized in Equation 6.1 below:

$$e_i = c_i + \Delta_i = c_i + a_i \times (|\Pi|-1) \times l^{\max} \tag{6.1}$$

Where e_i represents τ_i's WCET after considering the WCD effect, as defined by fTC.

6.2.2 pTC Contention Model

However, the fTC conditions are obviously too pessimistic and unlikely to happen in real scenarios. For that reason, in pTC, we bound the number of conflicting accesses, based on available information on tasks' access profiles. We consider actual accesses to better assess which contenders and accesses might cause conflicts and incur contention delays. Overall, by reducing the WCD, we obtain a consistent reduction in the WCET as only the possible contending accesses and types are considered in a given scenario.

In most of the scenarios, we do not need to assume that every access is conflicting with accesses from every contender (each causing an $l^{\max}$ delay). In fact, most of the accesses are distributed along tasks' execution in instants when no contenders are found. However, for the sake of safeness, when contention happens, we will still have to assume that our task under analysis is the latest one to be served. That will ultimately result in being able to safely reduce the considered maximum contention, leading to tighter task budgets.

The iterative approach aims at finding pTC WCD bounds. Given an initial set up (task WCET and overlapping), the technique consists in repeatedly applying the analysis until a fixed point is reached, that is the WCD bounds computed at iteration I_k did not change compared to iteration I_{k-1}.

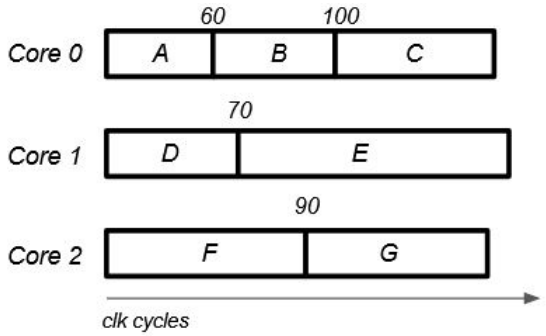


Figure 6.1: Overlapping

The following steps are performed at each iteration:

1. Join up the tasks' budgets (without any slack time in between) mapped to each core and "pair" their accesses in the most pessimistic manner. To do so, we first derive, for each task, the set of potential contenders across all other cores. For instance, as Figure 6.1 shows, task B is running together with D, E, F and G. Secondly, we start pairing B's accesses against the ones in each contender that incur the highest latency. If $a_B > a^t_{j \triangleright B}$ being t the type of access incurring l^{max}, we keep pairing remaining B's accesses with the accesses in the contender that cause the second-highest contention delay, and so on so forth.

2. Considering the analyzed access pairing, calculate the new budget inflation that these access pairing would imply, provided the latency of each access type can be obtained.

3. Update the tasks budget to form a (possible) new alignment scenario.

To perform these steps, a key point to analyze is how we initially allocate tasks to start performing the pairing analysis in the first iteration. Two main cases were studied: start joining up tasks considering their maximum execution time in isolation, and their fully time composable triggering times.

The next simple example illustrates that in fact, different starting points lead to different computed makespans. The example characteristics are as follows:

1. Only 2 cores are considered.

2. Only 1 type of request is assumed, incurring a maximum latency of 10 cycles.

3. Tasks A and B are allocated to Core0 (π_0), and C and D to Core1 (π_1).

For each task, we assume that the following information is available: core mapping, number of accesses and WCET in isolation (from which we also derive the fTC bound). Table 6.1 summarizes the data assumed in our example.

Task	**Accesses**	**Time Isolation [cycles]**	**WCET (fTC) [cycles]**
A $\rightarrow \pi_0$	4	60	100
B $\rightarrow \pi_0$	3	100	130
C $\rightarrow \pi_1$	2	70	90
D $\rightarrow \pi_1$	3	80	110

Table 6.1: Example input data

The WCET fTC times are calculated by applying Equation 6.1. Bearing in mind that in this example we count on only 2 cores, and that there is only one latency kind, the WCET of task A is calculated as: $60 + 4 \times 1 \times 10 = 100$. Analogously, the rest of WCETs are obtained.

6.3 Execution time with fTC as starting point

fTC is by definition the maximum contention a certain task under analysis can suffer, regardless of which contenders it encounters. It therefore represents a safe starting point for the analysis. Nevertheless, we want to understand whether different starting points will converge and lead to a common solution and, if they do not, whether the smaller bound is still safe.

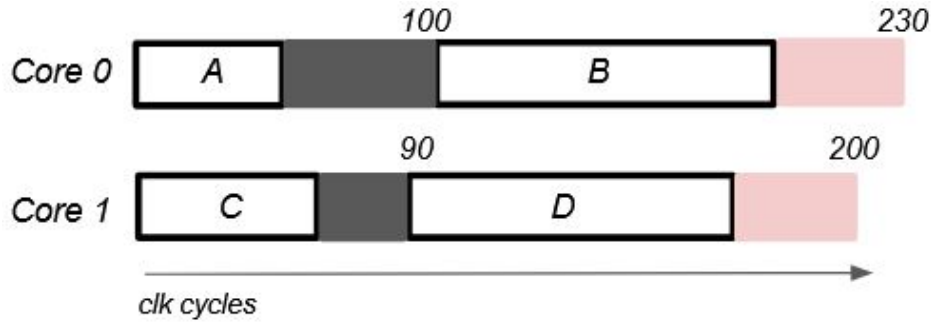


Figure 6.2: pTC from fTC - Iteration 1

Figure 6.2 displays the initial starting point if we consider fTC bounds. In that case, we start assuming the maximum contention and will keep reducing it by checking the actual tasks' alignment. Task τ_i is only triggered after the budget of task τ_{i-1} expires, which

boundary is marked with the stripped-lines area. In that scenario, we consider which tasks are overlappings with the task under consideration. Among the tasks that overlap, we have to pair their accesses in the most pessimistic way (as per Equation 5.3). That means, that we have to start assuming that each access of our task under analysis clashes with those in overlapping tasks, if any, that incur the longest latency. For the sake of simplicity, in this example only one type of access is considered.

Referencing the presented equations in Section 5.2, a certain task under analysis will only pair the minimum between its accesses and the contenders'. Moreover, a task under analysis can not collide more than its own accesses against tasks allocated to a same core. For instance, in Figure 6.2, D can at most pair 3 accesses globally to π_0. In contrast, from π_0's perspective, both A and B must account for the 3 accesses of D (over-accounting effect).

Bearing that in mind, we start pairing tasks accesses. We can appreciate that the WCET of task A (100 cycles) is paired with C and D from π_1. A is accessing the bus 4 times, hence can pair at most 4 accesses with a singular task, and at most 4 accesses within tasks of a contending core. This is a safe assumption since the contention effect on the task is the same no matter whether the clashing access occurs with task τ_j or with task τ_{j+1}. As a result, we sum the accesses of task C and D and take the minimum between that sum and A's accesses, that is, 4 contending accesses for A. Analogously, the rest of contending accesses per task is calculated.

A $\rightarrow$ Its 4 accesses are paired with either C or D ones (as they sum 5).

B $\rightarrow$ Task B performs 3 accesses. Since it only co-runs with D, which also accesses the bus 3 times, all B's accesses are paired.

C $\rightarrow$ This task only runs in parallel with task A from π_0, and since this contender performs more accesses, we can pair all C's accesses with that core.

D $\rightarrow$ We see that task D which counts on 3 accesses, and since it runs in parallel with A and B, it can clash its 3 accesses with either of them too.

The next step requires to realign the timing budget to the new contention bounds and update tasks' triggering time accordingly. The new budgets can be computed as the time in isolation of each task plus the calculated contention (pTC from now on). The triggering times of successor tasks, as always, is adjusted to previous task's budget. As it can be noted, the triggering time of a task is the summation of the budgets of the predecessor tasks. In this simple example, the alignment is exactly the same, and there is no need to reiterate the process (a fixed point has been reached). The computed budgets and triggering times define a safe schedule for the MIF. A further iteration would yield the exact same results.

Note that if the task pairs were the same but the release times had slightly been

Task	pTC_1	Triggering $Time_1$
A	100	0
B	130	100
C	90	0
D	110	90

Table 6.2: pTC after iteration 1 - fTC as starting point

modified, we would repeat a further iteration to recompute the budgets, as it could be the case that more or less accesses would collide as a result of this incremented or decremented budget.

Although these computed tasks' budgets and release times represent a safe solution, the fact of having started assuming the maximum contention delay per each task introduces some extra pessimism, which can be avoided by taking the opposite assumption (i.e., times in isolation with no contention) as a starting point, as will be made evident in the following Section.

6.4 Execution time with Isolation as Starting Point

On the other hand, the same exact process can be applied but varying the first initial budgets and consequently pairings.

Next, we present a counter-example showing how the makespans can be shortened if starting from time in isolation. The execution time of tasks in isolation and the number of accesses they perform are the same from the previous example.

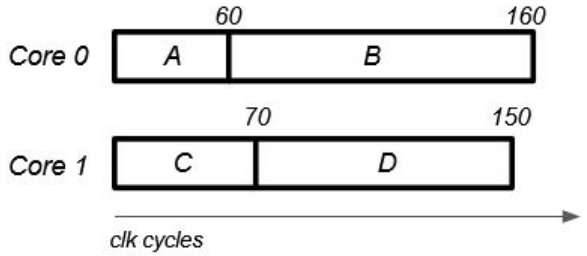


Figure 6.3: pTC from isolation time - Iteration 1

Similarly, we perform the core step consisting in pairing accesses:

In this case, the tasks' budgets keep increasing most of the times, instead of reducing as it used to happen when starting from the fTC bound. Table 6.4 summarizes the new budgets and release times after one iteration:

Since there has been a variation in tasks' budgets, we must proceed with another iteration, placing the tasks with their new computed budgets:

A → Only 2 of its 4 accesses are paired with the ones from C.
B → Since C and D co-run with B and they perform more accesses than B, B pairs all its accesses.
C → Analogously, C can pair its 2 accesses with A and/or B.
D → D and B run together and they both perform 3 accesses which are paired.

Table 6.3: Access pairings iteration 1 - Isolation as starting point

Task	**pTC$_1$**	**Triggering Time$_1$**
A	80	0
B	130	80
C	90	0
D	110	90

Table 6.4: pTC after iteration 1 - Isolation as starting point

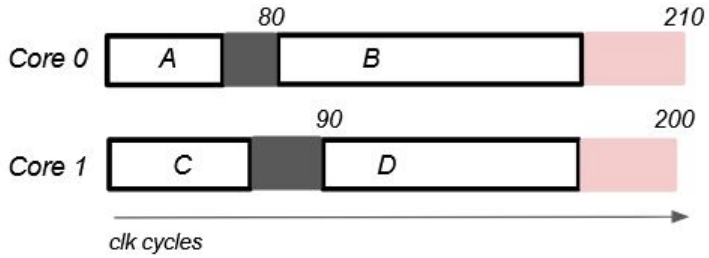


Figure 6.4: pTC from isolation time - Iteration 2

Despite budgets have been increased with respect to the previous iteration, the tasks' alignment has not been altered, and hence the pairing will be the same, leading to the same budgets calculated in iteration 1 (see Table 6.5).

Task	**pTC$_2$**	**Triggering Time$_2$**
A	80	0
B	130	80
C	90	0
D	110	90

Table 6.5: pTC after iteration 2 - Isolation as starting point

Once again, since $I_2 == I_1$, a fixed point has been reached, meaning that no further iterations would either add or remove contention delays.

6.5 Comparison, Benefits and Safeness of the Approach

Considering the overall makespans computed under the two methods, we observe that the makespan in π_0 is 20 cycles shorter when starting from isolation execution times. It is important to understand whether the lower makespan value can still be considered a safe bound, i.e. if a run-time alignment could possibly lead to a worse execution. This also corresponds in understanding whether, by starting from the fTC bound, we are introducing impossible scenarios.

The reason why starting from fTC is more pessimistic is that it includes unrealistic scenarios. In fact, in order for a certain task to be considered to be overlapping with another from another core, we have to take into account the effects of contention **without** considering the contention among these same tasks. Those cases where overlapping occurs only because we have already accounted for some degree of contention between the considered tasks, are definitely not feasible and can be safely discarded. Although the pessimism in this case was of only 20 cycles, it becomes clear that with far more accesses, different latencies, more tasks, and more cores, the unnecessary pessimism introduced would be much larger. Applying this reasoning to the above example, if we start aligning the tasks considering the fTC, task A would be paired with task D, accounting for additional contention. Nevertheless, this alignment is already considering the access pairings with D, which will never happen on a real case if A and C are triggered at the exact same cycle. Since C only performs two accesses to the bus, and A is only running in parallel with C, 20 cycles is the maximum contention delay that A can experience. As a conclusion, starting from the isolation execution times provides a tighter – and still safe – result.

In any case, we still have to justify why the resulting computed WCD with this method is indeed safe. We could intuitively suspect that it could be the case that by not pairing the maximum number of accesses at each moment as we currently do, could lead to a larger system-level makespan. Even though, with this approach this can not happen.

To clarify that, let's see the previous example with a variation in tasks' accesses such as Table 6.6 denotes.

Task	**Accesses**	**Time Isolation [cycles]**	**WCET (fTC) [cycles]**
A $\rightarrow \pi_0$	10	60	160
B $\rightarrow \pi_0$	4	130	170
C $\rightarrow \pi_1$	2	70	90
D $\rightarrow \pi_1$	8	120	200

Table 6.6: Sample input data: safety justification

We could at first glance think, that if A is the latter one to get granted accesses to the bus in case of a collision with C, that would increase its budget by 20 cycles whereas

C would finish its execution within 70 cycles, provoking A to run in parallel with D as well, and hence pairing its remaining 8 accesses with D (and as a result incurring a larger makespan). However, that case is excluded with this method, since we are *forcing* the release times of tasks statically, so task D will be triggered at cycle 90 (91 more precisely, after C's calculated budget), making it impossible for task A to ever face task D, because task A completes at cycle 80. Here lies the main point of this approach: the real tasks alignment on a platform would be the one we are statically enforcing based on the analysis results. Under the non-work-conserving assumption, when a job terminates before its allocated budget, the next job of the subsequent task in the same core, will still not be triggering until the statically scheduled time, guaranteeing that no pairing scenario can happen that was not considered in the analysis.

6.6 Experimental Framework and Setup

After the description of how we obtained the bus event counts and latencies for the LEON4 platform, and before entering into the actual evaluation of the proposed models for the WCD computation, we describe the experimental framework and setup used in the evaluation itself.

The experiments we conducted mainly consisted in comparing different techniques to compute the WCD for a task set and under a given contention scenario, with the goal to assess how the overall makespan may vary depending on the adopted approach. When not otherwise specified, the analysis is always applied within the scope of a single scheduling slot (MIF) as it is at its boundaries that timing budgets must be enforced. The analysis can be straightforwardly extended to fit the hyperperiod (MAF) boundaries by applying it to each MIF individually.

Experiments were performed on a large number of tasks sets. We assumed a fixed scheduling slot per MIF of 100 milliseconds, which is a representative value for statically scheduled systems [56], to allow defining experiments over varying overall MIF-level utilizations (computed based on tasks' timing requirements in isolation). However, any other slot size can be considered and the same procedure applies. Task sets were randomly generated using the UUniFast algorithm [17]. In particular, 4,000 task sets were generated for each utilization threshold in the range [0.1, 1] with 0.05 steps, summing up a total of 76,000 task sets. The number of tasks produced per task set to fit in each utilization threshold was up to 8. As an example, willing to generate an experiment with a MIF utilization of 50%, the UUniFast algorithm would be used to generate a random number of tasks, so that the sum of their execution time in isolation would be equivalent to the utilization of the MIF (50ms in this very case).

From the standpoint of the inter-core interference, the amount of bus activity is a critical aspect of the problem. The tightness and its improvement with respect to other techniques may vary depending on much memory-intensive the task sets under analysis

are. In order to warrant a fair evaluation, some access profiles were derived, based on the memory access and cache statistics of benchmarks in the EEMBC [65] and mediabench [54] suites. Accesses and L2 misses *per kilo (thousand) instructions* are considered, which are abbreviated into APKI and MPKI respectively. For each generated task set, four variants were produced, where tasks were characterized according to the access profiles summarized in Table 8.7. Profiles range from the CPU-bound profile (*CPU*), relatively robust in terms of contention, to the MEM- and BUS-bound profile (*B+M*), which instead are prone to consistent contention effects. Task's bus accesses are determined proportionally to the number of instructions in the task itself.

Profile	Description
CPU	$\leq$ 75 APKI $\leq$ 1 MPKI
BUS	$>$ 75 APKI $\leq$ 1 MPKI
MEM	$\leq$ 75 APKI $>$ 1 MPKI
B+M	$>$ 75 APKI $>$ 1 MPKI

Table 6.7: Categories created from per-access profiles

Both the ILP and the iterative method models are populated with the data from the task specification and access profiles. Experiments are analogously repeated for all access profiles. For a given utilization threshold, each experiment consists in selecting 4 task sets (one per core in the LEON4): one task set is selected as the focus of the analysis, whereas the 3 remaining sets are assumed to be mapped to the contending cores. The original task set makespan is compared against the one obtained by factoring in the WCD for each task in the set. In particular, what is interesting to be observed here, is the ratio of task sets whose makespan does not overrun the slot boundaries after accounting for the WCD. The tighter the WCD bounds the smaller the increase in the makespan and the lower the probability of overruns.

These solutions have been compared against similar approaches in the state-of-the-art, as will be detailed in the description of the individual experiments. Each experiment is specifically designed to underline an aspect that differentiates the proposed approach from the state-of-the-art.

It is important to emphasize that the inputs required to run the experiments are quite limited. The only inputs required by both approaches are: (i) the number of cycles that tasks take to execute when run in isolation; and (ii) the number of each tracked bus access event, obtained via PMCs readings. All inputs can be derived from the tasks running in isolation (without contention effects).

It is assumed that the number and types of accesses can be derived either by static analysis [76] or by collecting information from the performance monitoring counters, similarly to [21]. The latter approach, which was presented in the previous section, was used in our proof of concept evaluation in Section 7.5. Each access type is characterized by a

worst-case latency. In the upcoming experiments, the per-type latencies, provided as an input to the model, are those described and obtained from the LEON4 board. Similar experiments could be derived for any other access types and latencies.

6.7 Results: Iterative Approach

In the following set of experiments, the tightness of the iterative approach is evaluated. Since it does not rely on any assumption on the application semantics (e.g., separated memory and computation phases) or specific support from the underlying hardware or RTOS, it is first assessed against approaches with similar and comparable assumptions [73, 22, 63]. However, for the sake of completeness, the evaluation also considers a more restrictive computational model where tasks execution is divided into phases, to assess the flexibility of the technique to adapt to and support different scenarios.

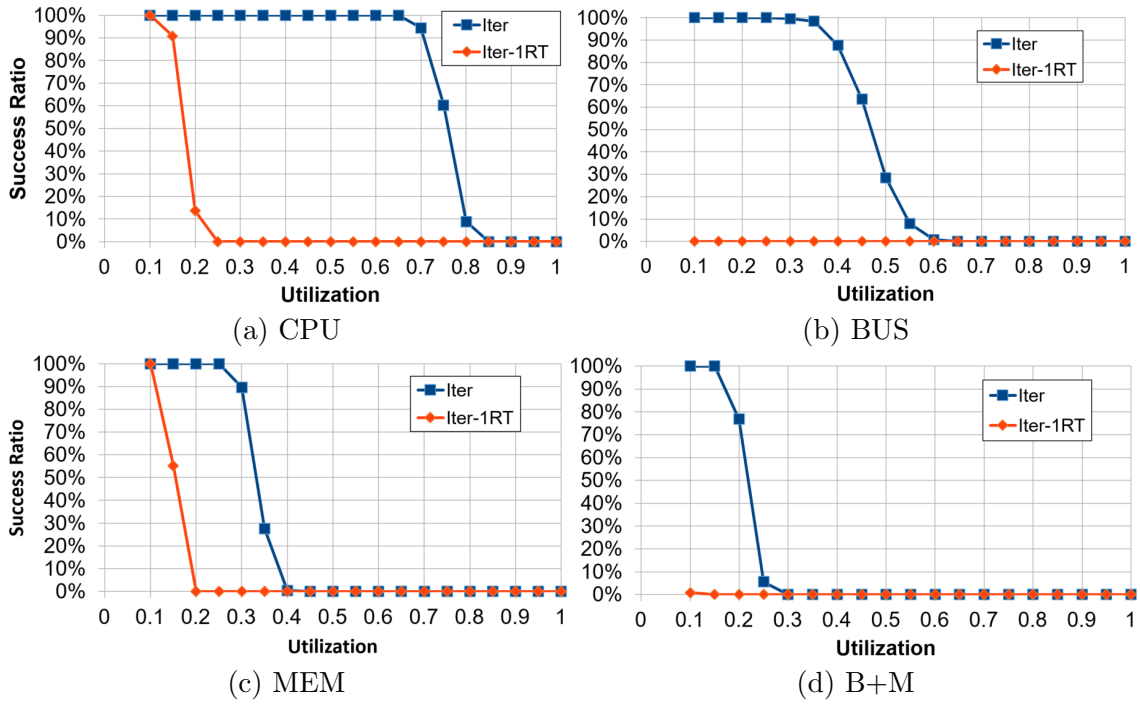
All the experiments of this approach were run on a laptop system, consisting of a 4x Intel i7-5600U CPU running at 2.60GHz, with 16GB of RAM memory. The approach is implemented by means of a python script, which executes and produces the computed result within a negligible time.

6.7.1 Multiple access types distinction

The first case we wanted to analyzed, is the benefits that can be obtained by distinguishing several types of access in the WCD computation. The approach in [22] assumes a single request type, which is equivalent to assuming that all requests incur the longest (worst) latency, as a safe WCD bound must to be provided. The amount of pessimism stemming from this limitation is platform dependent. In relation to our case and platform of choice, this approach is forcing to assume that all requests take 31 cycles (dirty misses) as shown in Table 5.2.

In the next experiments, we compare our iterative approach against its variant considering just one access type, as in [22]. Both approaches are evaluated by observing the ratio of task sets whose timing requirements end up overrunning the schedule slot, when the WCD is added to their timing budget computed in isolation. Figure 6.5 shows the success ratio obtained for each utilization threshold, and all described workload types (CPU, BUS, MEM, and B+M). Each sample consisted in applying the WCD analysis to 1,000 randomly generated task sets (as explained in Section 6.6). The results of the iterative approach are referenced to as *"Iter"*, whilst the iterative approach only considering a unique request type of bus access is referred to as *"Iter-1RT"*.

Figure 6.5 clearly reflects the benefit brought by being able to distinguish and track different types of accesses. As expected, under all access profiles, *Iter* outperforms *Iter-1RT*. It is interesting to observe that, for memory intensive access profiles, not differentiating between request types results in not being able to allocate tasks for even a 10% of MIF utilization. This confirms how important it can be to be consider bus access types to avoid excessive pessimism in the WCD computation.



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 6.5: Iterative vs *Iterative-1RT*

It is worth noting that the utilization threshold refers to the ratio between slot size and tasks WCET in isolation, i.e. not factoring in multicore contention. As a result, as the pressure on the shared resources increases – which mainly happens for MEM and B+M workloads, given that dirty misses and hence accessing the memory is the event which incurs the highest latency – the resulting multicore CPU utilization goes beyond 100% (at core level) simply making the task set not schedulable any more.

6.7.2 Multiple execution phases distinction

To complete the assessment of the suggested approach, we can also experiment on the flexibility of this approach by extending the underlying model to represent execution phases within tasks. Several studies [64, 63, 7, 15, 18] assume application semantics that clearly separate phases dedicated to data exchange (E) (usually from/to a local on-chip memory) to other devoted to pure computation (C). For instance, it is common that a certain application or program loads data in the first place, then operates and executes on that data, and finally stores back variables and the results of these operations. Phases are usually exploited to devise scheduling solutions aiming at conflict avoidance, but they also naturally constrain the possible task overlappings and, hence, access pairing. We are not interested here in imposing any scheduling restriction, instead, in evaluating the reduction in WCD that can be achieved under a favorable – and more restrictive – scenario,

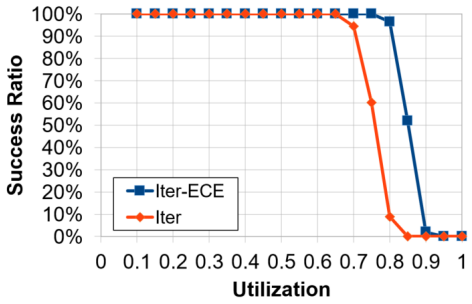


Figure 6.6: Iterative vs *Iterative-ECE* CPU profile

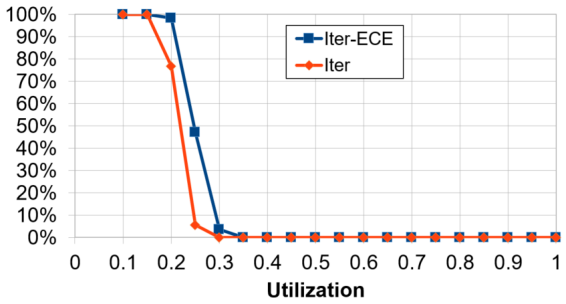


Figure 6.7: Iterative vs *Iterative-ECE* B+M profile

where additional details are available. The baseline of the iterative approach against its adaptation that supports a scenario where tasks are split into three phases is as follows: a relatively large computation phase is preceded and followed by exchange phases, with accesses to shared resources exclusively occurring during the latter. In an exemplary fashion, we can assume a uniform and fixed duration of each phase in the proportion of 20% and 60% of the task execution budget, for the E phases and the C phase respectively. Tasks carry out half of their accesses in each E phase. The evaluation in this case is only restricted to the workloads with highest and lowest pressure on shared resources, i.e. *B+M* and *CPU* respectively. These profiles are already significant enough and extrapolable conclusions can be derived for the intermediate access profiles described.

As expected, splitting tasks into partitioned phases of code where memory exchanges happen, allows for lower WCD bounds. The WCD reduction is entirely ascribable to the restrictive assumptions on access distribution. The separation into phases rules out several pairing scenarios that are instead necessarily considered when accesses can happen according to any possible distribution within the task execution, leading in that way to better results.

Chapter 7

ILP Approach

In this section, we present the ILP (Integer Linear Programming) approach. We begin by discussing its foundational principles. Following this, we will outline the specific constraints and objective function that form the core of the ILP model. These constraints encapsulate the timing requirements and resource limitations inherent in the discussed architecture. Additionally, we will enumerate the necessary assumptions for applying this approach effectively. Finally, we will demonstrate the robustness and performance of the ILP approach through a detailed analysis and showcase some experimental results to validate its effectiveness.

7.1 System-Level Guarantees: The ILP Approach

In this approach, alternatively to the iterative one, we are interested in computing tight **system-level** WCD bounds (makespan of the MIF functions) as a means to increase the guaranteed performance on each MIF, and hence to be able to accommodate more functions while preserving the timing constraints. In contrast to its task-level counterpart, the system-level WCD computation implies considering the WCD over a sequence of tasks (under the non-preemptive assumption, common in cyclic executive systems). This allows discarding the possibility of considering an interfering access to cause conflicts on more than one task in the same core.

Since in this case we are performing a system-level analysis, the problem gets a bit more complicated, since as we have seen in a previous example, a locally bad alignment does not necessarily bring us to the worst case execution time. To that end, the only solution is to analyze all the possible alignment scenarios and compute the most unfavorable one to ensure safety in a brute-force manner. Doing so in bare code would be computationally expensive, and for that reason, an Integer Linear Programming was chosen. The reason of that choice, is that ultimately, what we need to obtain is the number of cycles which we need to book to ensure that tasks can execute safely. This, can be modelled as an integer value, and so the number and combination of accesses paired to reach that number of execution cycles.

ILP consists of 4 phases:

1. Define variables → we define the variables for which we want to find an optimal value (e.g. number of dirty L2 misses from τ_j delaying τ_i).
2. Define constraints → limiting the value of the variables by means of the presented equations.
3. Specify objective → we must identify which result we want to optimize (e.g. maximize the makespan).
4. Solve problem → run one of the many existing solvers to solve our problem.

In order to model all that, we have set up the problem in Python. An interface library named PuLP serves as an intermediate layer for one of the solvers to translate the programming language as Figure 7.1 depicts. Several solvers were tested, being IBM's CPLEX the one which calculated a solution the fastest.

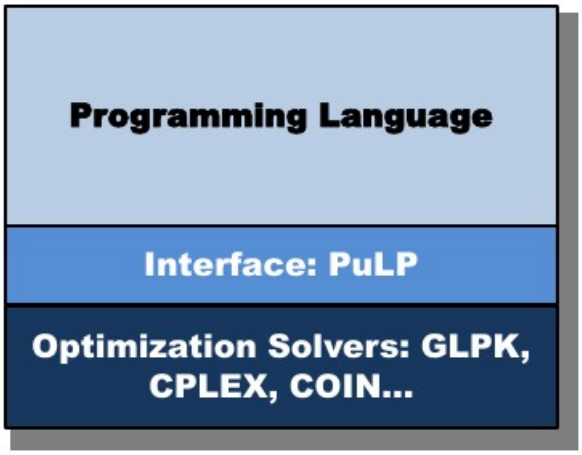


Figure 7.1: ILP layers

To properly capture all the system-level requirements, we have to add further constraints to our model which will be detailed below.

7.2 ILP Constraints

In the context of ILP, constraints are essential components that define the limitations and requirements of the system to optimize. These constraints ensure that the model adheres to the necessary timing and resource usage criteria, enabling accurate and reliable computation of worst-case delay bounds. By incorporating these constraints, we can precisely model the interactions and dependencies among tasks, thereby providing a comprehensive

framework for analyzing multicore systems. The following formulation formally describes the constraints that have been added.

7.2.1 Bounds on Task-to-Task Access Pairing

In contradistinction to the iterative approach, in which we are setting the triggering times and hence must add the aforementioned sources of pessimism at task-level, in this approach we do can consider that in fact only one of the tasks will access the shared resource the latest, in the event that more than one are trying to access it at the same instant. For instance, if tasks τ_i and τ_j (assumed to be allocated to different cores and running in parallel) both perform 10 accesses and they happen to perform them all at the same instants, in the iterative approach we would have to consider that **both** can be contended by 10 $\times l^t$, whereas in ILP we can model that this contention is the overall contention in both (as either one of them will be the first one accessing it, i.e. not incurring any delay).

More precisely, task pair-wise interference (i.e. the sum of the paired accesses on one direction or the other) cannot be greater than the access counts in both tasks:

$$a_{i \triangleright j} + a_{j \triangleright i} \leq \min(a_i, a_j) \tag{7.1}$$

7.2.2 Bounds on Core-to-Core Access Pairing

While the above and the previously defined task-level constraints are fundamental blocks in the model, they still fail to accurately model the way accesses are paired from a system perspective. The system level perspective considers pairings over the set (sequences) of tasks mapped to a core. Core-level constraints allow modeling the fact that one specific access of a given task cannot be delayed (i.e., be paired) with more than one conflicting access per each contender core. For example, a given task access cannot suffer contention from two accesses performed by two distinct tasks, mapped to the same contender core. These constraints are considerably improving the tightness of the computed WCD, and are one of the aspects that differentiate this approach from related works [72, 57, 25].

The constraint is formulated first from the standpoint of the interfering task, observing that any of its accesses cannot be paired multiple times with tasks on the same interfered core (it can be paired with at most one access per core). Accordingly, a constraint must be added for the sum of (interfering) accesses of an interfering task τ_j, that are *paired* with accesses of tasks on a given core $\pi_s \nleftrightarrow \tau_j$, not to exceed the number of accesses in τ_j itself:

$$\sum_{i:\tau_j \in \bowtie_i(\mathcal{T}_s)} a_{j \triangleright i} \leq a_j \tag{7.2}$$

Dually, we also have to enforce that the number of accesses paired with accesses triggered by the set of overlapping tasks from the same interfering processor can never

exceed the number of accesses in the interfered task τ_i:

$$\sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} a_{j \triangleright i} \leq a_i \tag{7.3}$$

In a scenario admitting different types of requests (and latencies) as it is the case, we can redefine the constraints by modeling also access types as a further dimension to the problem, as done for the task-to-task constraints. The number of *paired* accesses of a given type, triggered by a given contender (τ_j) on the overlapping tasks on a given core cannot exceed the number of accesses of that type in the contender itself.

$$\sum_{i:\tau_j \in \bowtie_i(\mathcal{T}_s)} a^t_{j \triangleright i} \leq a^t_j \tag{7.4}$$

In its dual formulation, the number of *paired* accesses of a given type and triggered by the set of overlapping contender tasks on a given processor, is bounded by the overall number of accesses of any type in the interfered task (τ_i):

$$\sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} a^t_{j \triangleright i} \leq a_i \tag{7.5}$$

Generalizing the constraint in Eq. 7.5, we can also assert that the number of *paired* accesses **of any type** and triggered by tasks on a given processor is still bounded by the number of accesses of the interfered task itself:

$$\sum_{t \in \Gamma} \sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} a^t_{j \triangleright i} \leq a_i \tag{7.6}$$

It is worth noting that a necessary conservative assumption in the analysis of the WCD *at task level* would be to assume that the task under analysis always suffers the worst-case possible contention. As a consequence, pairing in the presence of typed accesses should conservatively pair those accesses that incur higher-latency first (see Eq. 5.3, 7.6). However, we do not need to model this constraint as the worst-case pairing (of a request of type t) to each task request is already induced by maximizing the overall *makespan*, as part of the objective function.

7.2.3 Model Constraints: Modeling Task Overlapping

We need to include in the ILP model a convenient definition of the overlapping condition between tasks. For each scheduling interval, the static schedule defines the set of tasks running on each core, which in turn identifies the set of possible task overlappings. At any instant, each task will only execute in parallel with at most one task per contending

core. Further, since tasks follow statically-defined precedence constraints, only a subset of overlappings are possible in practice.

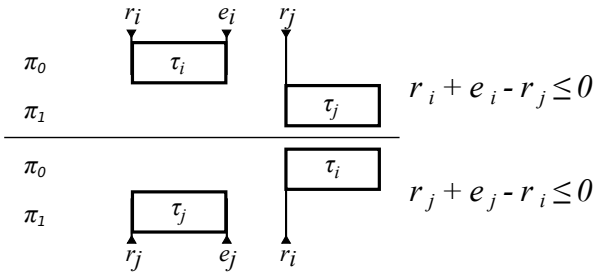


Figure 7.2: Non-overlapping tasks

The ILP formulation aims at identifying the worst-case overlapping among the feasible ones. In order to do so, we need to define an overlapping condition. We can formalize the negative sufficient condition for task τ_j overlapping with τ_i as follows:

$$\tau_j \notin \bowtie_i(\mathcal{T}_s) \iff (r_i + e_i \leq r_j) \vee (r_j + e_j \leq r_i) \tag{7.7}$$

That is, τ_i ends in the worst case no later than τ_j release instant or vice versa, as illustrated in Figure 7.2. We can also observe that Eq. 7.7 is a reflexive relation, so that $\tau_j \in \bowtie_i(\mathcal{T}_s) \iff \tau_i \in \bowtie_j(\mathcal{T}_{s'})$. Moreover a task cannot overlap with tasks mapped to the same core: $\tau_i \in \mathcal{T}_s \Rightarrow \bowtie_i(\mathcal{T}_s) = \emptyset$.

The overlapping condition, however, has to be *linearized* in order to be included in the ILP model. Modeling conditionals is relatively simple when modeling binary variables as conditions can be easily obtained by using boolean algebra, which unfortunately did not apply to this case.

For each pair of potentially overlapping tasks τ_i, τ_j we can model the overlapping condition by encoding a constraint on $\Delta_{j \triangleright i}$ (equivalent to using $\Delta_{i \triangleright j}$), which is set to be null when $\tau_j \notin \bowtie_i(\mathcal{T}_s)$ (and $\tau_i \mapsto \pi_s$). To this extent, a pair of auxiliary boolean variables $B_{i,j}, B'_{i,j}$ is used, as well as a large-enough constant (which will be noted as M), which is enough when set to an order of hundred thousands.

The condition that needs to be modeled also builds on integer variables e_i, e_j and r_i, r_j that represent tasks' inflated execution time budget and release instant respectively. When overlapping occurs, are affected by the contention effect on the tasks themselves and their predecessors in the scheduling slot. In fact, the time in isolation and the WCD will determine the budget allocated to each task in the slot. Given a task τ_i and the triggering time of its scheduling slot t_{MIF}, we can define a recursive relation to compute the release time of a task, where $pred(i)$ is the predecessor of τ_i, executing right before τ_i on the same core.

$$r_i = \begin{cases} t_{MIF} & \text{if } \tau_i \text{ is the first task in its MIF} \\ r_{pred(i)} + e_{pred(i)} & \text{otherwise} \end{cases}$$

Note that e_i represents the execution time budget assigned to τ_i including the WCD interference (i.e., $e_i = c_i + \Delta_i$). The interval $r_i + e_i$ is thus irrevocably reserved for τ_i execution (as per not work-conserving assumption).

Using this relation, we are able to encode the overlapping condition as a pair of constraints (one for each operand in the disjunction in Eq.7.7) as follows:

$$\Delta_{j \triangleright i} \leq 0 + M * (1 - B_{i,j}) \tag{7.8}$$
$$\Delta_{j \triangleright i} \leq (r_i + c_i - r_j) * \max(a_i, a_j) * l^{\max} + M * B_{i,j} \tag{7.9}$$

and

$$\Delta_{j \triangleright i} \leq 0 + M * (1 - B'_{i,j}) \tag{7.10}$$
$$\Delta_{j \triangleright i} \leq (r_j + c_j - r_i) * \max(a_i, a_j) * l^{\max} + M * B'_{i,j} \tag{7.11}$$

The correctness of the above formulation is shown by cases, focusing on the constraints modeling the first operand (Eq. 7.8 - 7.9) as the same reasoning applies to the dual operand. It can be recalled that $r_i + c_i - r_j \leq 0$ meets the not overlapping condition.

Case $r_i + c_i - r_j < 0$:

If τ_i and τ_j are not overlapping $r_i + c_i - r_j$ will take a negative value. Under all scenarios, the ILP solver will strive to maximize the bounds on $\Delta_{j \triangleright i}$ (i.e., having looser bounds). Therefore, for example, if $r_i + c_i - r_j < 0$, the solver will set $B_{i,j} = 1$ (thus activating the big M), to avoid $\Delta_{j \triangleright i}$ being upper-bounded by a negative number in Eq. 7.9. As a side effect, Eq. 7.8 will set $\Delta_{j \triangleright i} \leq 0$, which is exactly what we wanted to model.

Case $r_i + c_i - r_j = 0$:

In this case the tasks are not overlapping either. In fact, $\Delta_{j \triangleright i} \leq 0$ for any choice of $B_{i,j}$.

Case $r_i + c_i - r_j > 0$:

In this case tasks might be overlapping (depending on the dual conditions in Eq. 7.10 - 7.11). The solver will set $B_{i,j} = 0$, to avoid $\Delta_{j \triangleright i}$ being upper-bounded by 0 in Eq. 7.8. By setting $B_{i,j} = 0$, Eq. 7.9 will simply bound the variable $\Delta_{j \triangleright i}$ to a quantity that is *in all cases* larger than the theoretical maximum contention delay. In fact, a task cannot suffer more collisions than the number of its requests (a_i) or the requests from the contender (a_j), and for each colliding request it cannot incur an additional latency larger than the maximum latency for any request type ($l^{\max}$). That is, $\Delta_{j \triangleright i} \leq x \times \max(a_i, a_j) \times l^{\max}$ is a tautology for any value of x, and will be simply superseded (discarded) by the constraints on access pairing.

7.3 Objective Function

The objective function in an ILP model is a mathematical expression that defines the goal of the optimization problem. Typically, it is designed to either maximize or minimize a particular quantity, representing the criteria that need to be optimized.

The objective function is subject to the constraints of the ILP model, which define the feasible region within which the solution must lie. By solving the ILP problem, the objective function helps to find the best possible solution that satisfies all the constraints while optimizing the desired objective.

For a given task τ_i, Δ_i is defined as the WCD suffered by that tasks when executed within the considered MIF. The inflated execution-time budget reserved for τ_i in multicore execution is denoted $e_i = c_i + \Delta_i$. Consequently, the objective function in our case consists in maximizing the makespan of the set of tasks mapped to a core and executing within a given scheduling interval (MIF). This is equivalent, in terms of maximization, to the overall cumulative effect of contention suffered by those tasks. For example, considering core π_s (and a MIF):

$$\max makespan_s \equiv \max \sum_{\tau_i \mapsto \pi_s} e_i \equiv \max \sum_{\tau_i \mapsto \pi_s} (c_i + \Delta_i) \tag{7.12}$$

An alternative objective function would be that of maximizing the cumulative makespan across all cores within a MIF, such as $\max \sum_{\pi_s \in \Pi} \sum_{\tau_i \mapsto \pi_s} e_i$. However, this formulation would only provide realistic makespan figures, which cannot be considered as valid upper bounds for the individual cores.

The term Δ_i, used in the objective function to indicate the WCD suffered by τ_i, is the cumulative result of the contention caused by (paired) accesses from overlapping tasks mapped to contender (interfering) cores. For each task τ_i, $\bowtie_i(\mathcal{T}_s)$ identifies the set of tasks mapped to (interfering) core $\pi_s \nleftrightarrow \tau_i$ that can overlap in time with τ_i. At a task-to-task level, the variable of interest is $\Delta_{j \triangleright i}$ to represent the interference suffered by τ_i because of accesses triggered by $\tau_j \in \bowtie_i(\mathcal{T}_s)$.

The interference depends on the number *and* types of accesses in τ_j that are assumed to collide with accesses in τ_i. In order to model the different latencies entailed by multiple access types, $\Delta_{j \triangleright i}$ is defined as follows:

$$\Delta_{j \triangleright i} \stackrel{\text{def}}{=} \sum_{t \in \Gamma} a^t_{j \triangleright i} \times l^t \tag{7.13}$$

where $a^t_{j \triangleright i}$ represents the number of accesses of type t in τ_j that are *paired* with accesses in τ_i, and l^t represent the maximum latency for that type of access.

In fact, either a task is causing interference or is suffering from it. This constraint, however, will be automatically invalidated by maximizing the contention on one of the two tasks (i.e., setting one term on the left side of Eq. 7.1 to 0).

Under these conditions, pairing should conservatively assign higher-latency accesses first. This is not explicitly modeled as the worst-case pairing (of a request of type t) to each task request is already induced by maximizing the overall *makespan*, as part of the objective function.

Accumulating the interference generated by all overlapping tasks in all contender (interfering) cores:

$$\Delta_i = \sum_{\mathcal{T}_s:\pi_s \not\perp\!\!\!\perp \tau_i} \sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} \Delta_{j \triangleright i}$$

7.4 Results

In this section, the evaluation of the ILP formulation is presented. This multicore contention analysis (named ILP-WCD from now on), focuses on computing system-level WCD bounds rather than task-level ones, as a means to minimize the worst-case *makespan* within each MIF boundary, while providing strong performance guarantees. A shorter (guaranteed) makespan allows, in fact, to safely accommodate more functions within the same schedule slot.

Similarly to the previous experiments, a set of experiments has been analogously repeated for the ILP-WCD, in which the cumulative effect on the MIF makespan of the WCDs computed using this ILP formulation is compared against those obtained with comparable state-of-the-art approaches. The set of inputs employed is the same for each case with respect to the iterative approach.

As opposed to the iterative script, these experiments required considerably more time. For that reason, experiments were in this case executed on a cluster borrowed by the *Universitat Politecnica de Catalunya* (UPC), consisting of a 2x Intel Xeon E5-2630L v4 running at 2.2GHz, with 128GB of RAM memory. Nonetheless, after several improvements and refinements, *ILP-WCD* took only 2 minutes in the average case to compute the cumulative WCD for a given core, using the IBM CPLEX solver [2]. The experimental framework implements a fully-automated generation of the ILP inputs, based on the task set descriptions. Experiments were conducted by running utilization batches in parallel for each given utilization and task set access profile, so the computation process could be significantly reduced.

7.4.1 ILP-Specific Generation Process

The evaluation of the ILP-based method required additional steps with respect to the iterative approach. ILP-WCD requires to define an input-dependent ILP model for each experiment. It would not be reasonable to manually encode the ILP model for each experiment, considering the vast amount of tests that had to be run. We automated the experimental framework by developing a *code generator* script (programmed in Python) that

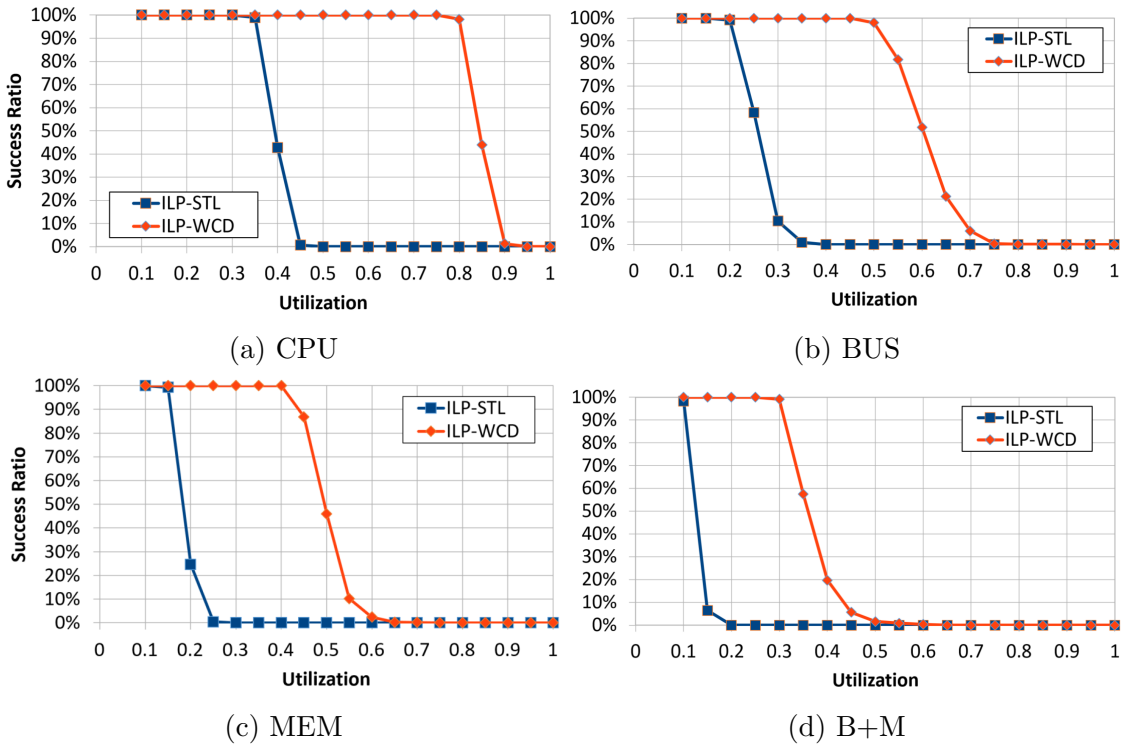
automatically generates the required code to be executed with the IBM CPLEX solver [2], provided the inputs of each core in a CSV format are given (randomly generated, in this case). Just to give an idea of the magnitude of an ILP model, only one single experiment with an average of 7 tasks per core would require around 15,000 lines of ILP-specific code for the makespan to be calculated. The implemented code generator sets the experiment up in less than a second. The number of lines of code grows exponentially as we add tasks since all the possible dependencies among tasks have to be encoded. What is more, as it will be noted in the upcoming experiments, the effect of splitting tasks into phases is also analyzed, hence the number of inputs further increases, causing the lines of ILP code to easily reach a magnitude of hundreds of thousands. For that reason, automating this process was of paramount importance.

Once the required ILP code is produced, it can be provided as an input to the ILP solver to calculate the optimal solution (makespan maximization of the core under analysis). The resulting output consists of the maximum number of cycles the makespan could take, and the required collisions of tasks accesses and consequently budgets of each task individually. All this process was automated to produce the thousand of experiments for each access profile and utilization.

7.4.2 Focusing on System-Level Bounds

The ILP-based WCD formulation proceeds by pairing accesses at system-level, across job boundaries, by considering the sequence of tasks executing in a core. This allows to capture the core-to-core pairing constraints introduced in Section 7.2.2. Mainly, the model benefits with respect to the iterative approach from the fact that we are preventing the accesses of a task to interfere with more than one access on the same interfered core (it can be paired with at most one access per core). To evaluate this aspect, *ILP-WCD* is compared against the baseline approach presented in [73], which instead operates exclusively at task level, without considering the actual contending tasks. The approach in [73] assumes that all accesses in a task can be paired with the accesses of all tasks in each contender core, regardless of the concrete task overlapping actually occurring at run-time. Basically, the approach only exploits task-level constraints and does not consider overlapping conditions. To perform the evaluation, the ILP model was briefly adapted to mimic [73] in this particular setting by disregarding task overlapping and core-to-core constraints. The implemented technique is referred to as *ILP single-task-level* (*ILP-STL*), as it mainly exploits detailed information on the task under analysis but uses only core-level cumulative information on the interfering tasks.

We can observe that *ILP-WCD* dominates *ILP-STL*, achieving good success ratios across all utilizations. In fact, the drop in success ratio ('knee' in the figure) for *ILP-STL* occurs in the utilization range $[0.15 - 0.25]$ across all workloads whereas for *ILP-STL* in the range $[0.35 - 0.85]$. Moreover, in terms of workloads, we can also notice that, as the pressure on the shared resources increases (from CPU to B+M) all approaches suffer a proportional reduction in success ratios, always with *ILP-WCD* improving over *ILP-STL*. The source of the pessimism of *ILP-STL* when compared to *ILP-WCD* stems from the



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 7.3: *ILP-WCD* vs. task-level interference (*ILP-STL*)

fact that the latter works at MIF (makespan) level, preventing that a given request from a given core is paired more than once. Instead, *ILP-STL* works at task level not keeping track of which requests from a given task have already been paired. This assumption is closer to the proposed iterative approach in that sense. As a result, in the worst case, one request of a task in a core can be paired up to K times, where K is the number of tasks in the considered contender core.

We highlight that this method has not been considered in the evaluation of the iterative approach as the latter already assumes a given task overlapping and identifies the set of potential contenders. Therefore, a comparison with an approach which not only considers all feasible alignments but also considers that any task in the system is a contender, would have been unfair and would not have provided additional insights on the method.

7.4.3 Multiple Access Types Distinction

Another interesting characteristic of *ILP-WCD* is that it captures the fact that the latency incurred by a conflicting access typically varies, even within the same resource, as it depends on the type of the interfering request. This proposed solution considers several request types and their associated latencies, as can be observed in all typed constraints in Chapter **??**. To evaluate this aspect, ILP-WCD is compared against the technique presented in [22], which improves over [73] by considering information on both, interfering

and interfered tasks, but does not consider that interconnects typically exhibit variable latencies, depending on access types. To conduct the evaluation, the ILP was modeled to mimic [22] in this particular setting. Analogously to the previous experiments, this technique will be referred to as *ILP with one-request-type* (*ILP-1RT*).

Again, we evaluate the ratio of task sets whose timing requirements do not exceed the schedule slot when the WCD is accounted for. Figure 7.4 shows the success ratio obtained for *ILP-WCD* and *ILP-1RT* for workload types and utilization thresholds. As expected, the *ILP-WCD* model outperforms *ILP-1RT* across all workload types and utilizations. The gap between the two curves identifies the increase in pessimism incurred by *ILP-1RT* with respect to *ILP-WCD*. Interestingly, the pessimism gap between *ILP-1RT* and *ILP-WCD* is more sensitive to the amount of bus requests rather than memory ones. This is explained by the fact that each request is assumed to take the longest latency: for bus requests, either loads or stores, that reach L2 but do not miss in the cache (hence not contributing to the MPKI), the incurred pessimism is larger than that associated to actual memory requests. In other words, *BUS* profiled tasks perform more accesses than *MEM* ones. The worst-case latency $l^{\max}$ in this platform is 31, which corresponds to the latency incurred in case of dirty L2 cache misses (either loads or stores). We can compare the overestimation incurred by each bus request as follows. For stores hits ($s2h$), the pessimism added is $l^{\max} - l^{s2h} = 31 - 1 = 30$ cycles, whereas for loads hits it results in $l^{\max} - l^{l2h} = 31 - 8 = 23$ cycles. Instead, *ILP-1RT* incurs notably less pessimism for L2 clean misses, where $l^{\max} - l^{s2mc} = 31 - 28 = 3$ (the same holds for l^{l2mc}). Finally, no additional pessimism is introduced on dirty misses, as $l^{s2md} = l^{l2md} = l^{\max}$. This behavior can be observed comparing Figure 7.4(b) for BUS and Figure 7.4(c) for MEM. In the former, tasks trigger a larger amount of bus requests than in the latter, resulting in reduced success ratio for *ILP-1RT*.

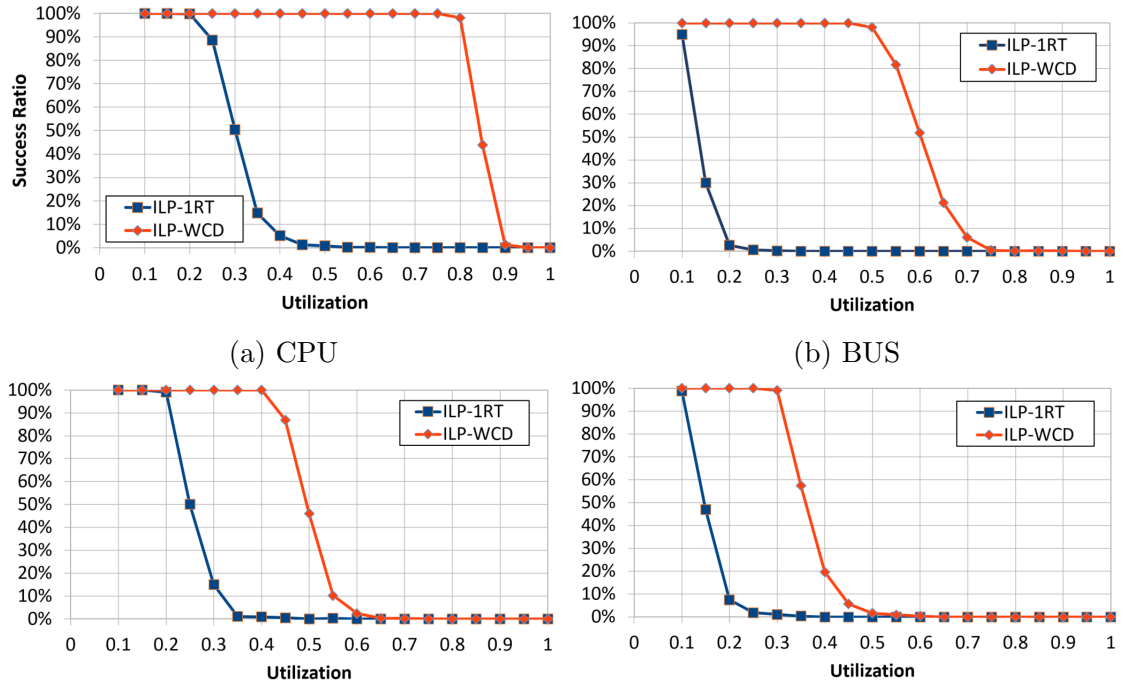
7.4.4 Multiple Execution Phases Distinction

Finally, it is also shown that as with the iterative approach, this approach is flexible enough to deal with phases of tasks to obtain an even tighter estimate.

As done for the previous experiments, ECE phases are assumed with a proportion of 20%, 60% and 20% respectively as well, assuming bursts of memory accesses at the beginning and end of each task.

Figure 7.5 compares the success ratio of *ILP-WCD* and *ILP-ECE*. As expected, *ILP-ECE* always outperforms *ILP-WCD*. However, the ECE execution model requires changes to the application and it is hard to apply on cache-based systems, where accesses to the bus cannot be scheduled at will.

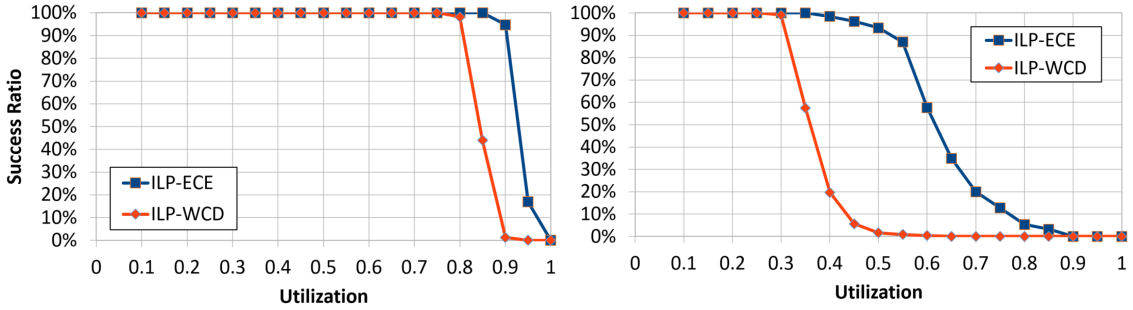
More importantly, this experiment demonstrated that the ILP-WCD formulation also supports it and is flexible enough to adequately model phases with any kind of accesses distribution. For the particular workloads and setup in this experiment, we can see that the improvement of *ILP-ECE* over *ILP-WCD* is considerably larger under the *B+M* workload. This is in line with the characterization of the workloads: *B+M* is meant to incur the highest contention effects and the advantage it takes from the *ECE* setting is proportional



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 7.4: *ILP-WCD* vs. single access type (*ILP-1RT*)



(a) CPU (b) B+M

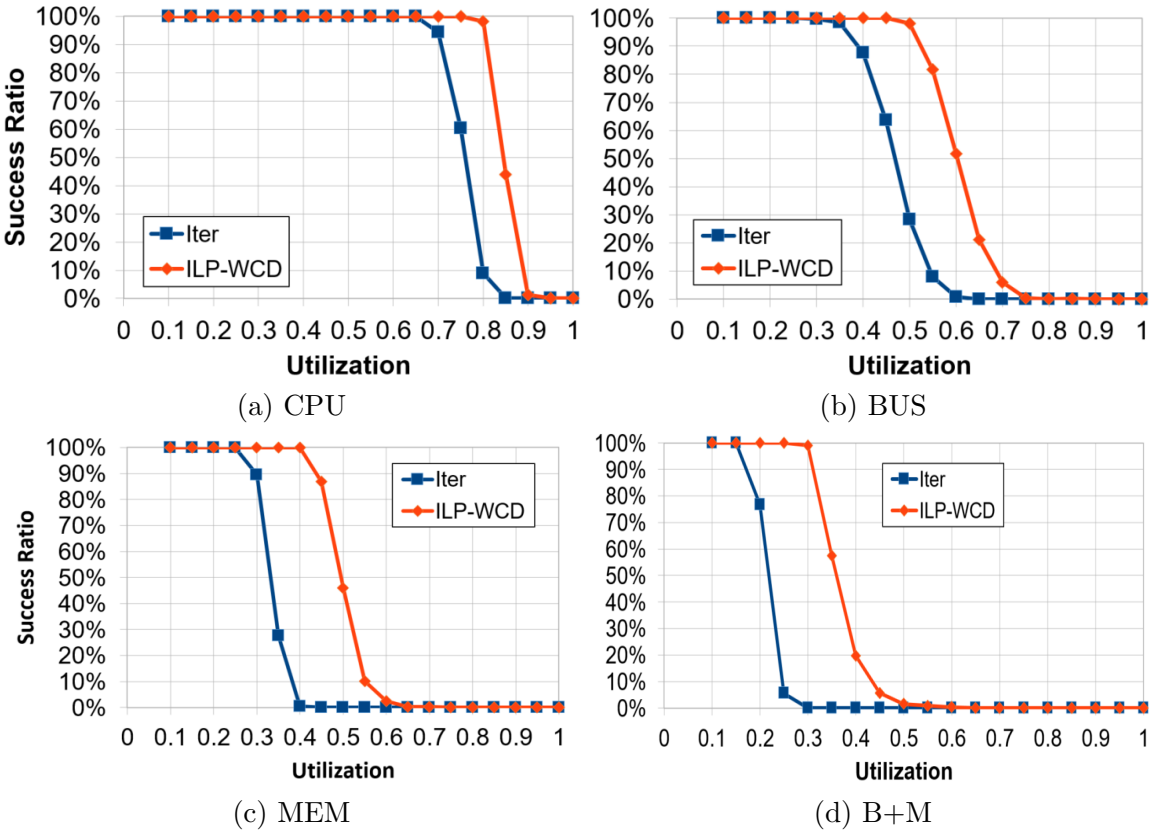
Figure 7.5: Comparison of *ILP-WCD* and *ILP-ECE*

to the number of accesses.

Comparison of the Proposed Approaches

Finally, we compare the two proposed approaches. Results are reported in Figure 7.6.

As expected, since the ILP method computes system-level bounds and avoids over-accounting of accesses, the obtained WCD bounds outperform those obtained with the iterative approach. It must be noted, however, that the iterative approach's main strength does not lay on the tightness of the overall cumulative bounds at system-level, but on the



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 7.6: Iterative vs. *ILP-WCD*

tasks' budget guarantees. As a result, it can not be as competitive as the ILP in that sense, which aims instead at providing tighter system-level bounds. However, despite the sources of pessimism that are inherently introduced in the iterative method, as detailed before in this chapter, it can be appreciated that it still provides reasonably tight results at significantly lower analysis time. The sources of pessimism in the iterative approach (and hence the slack time between tasks' activation), gets partially compensated by the fact that in ILP we are not discarding any overlapping scenario, which in turn can lead to extremely adverse overlapping and alignment for all tasks in a core.

It can also be noted, that the overall makespan computed differs as tasks' access profiles are higher in memory access (MEM and B+M) terms, which is explained by the fact that the over-accounted safely-reserved budget is higher per each task in the iterative approach (since the system experiences more dirty misses).

7.5 Proof-of-Concept Evaluation

At last, a proof-of-concept is provided. We consider practical applicability as a main concern: it was therefore important to provide evidence that the proposed methods are easily

applicable without relying on unrealistic assumptions. Furthermore, an evaluation on a real board served as a whole validation process, from extracting the PMCs during execution in isolation, to defining task set partitions and mapping them to different cores, to executing them under potential contention, and finally, to calculating safe WCD bounds. With respect to the latter aspect, we wanted to provide empirical evidence that real executions on a board can never exceed the calculated bounds (as this would imply that they do not provide safe guarantees) and the provided bounds are at the same time realistic (not exceeding in pessimism). So, in a nutshell, this experiment proves that given a set of tasks running on real hardware, both of the hereby proposed models end up computing a makespan that ensures that the reserved slot is enough to accommodate the timely execution of all applications.

In order to carry out this experiment, the following steps were followed:

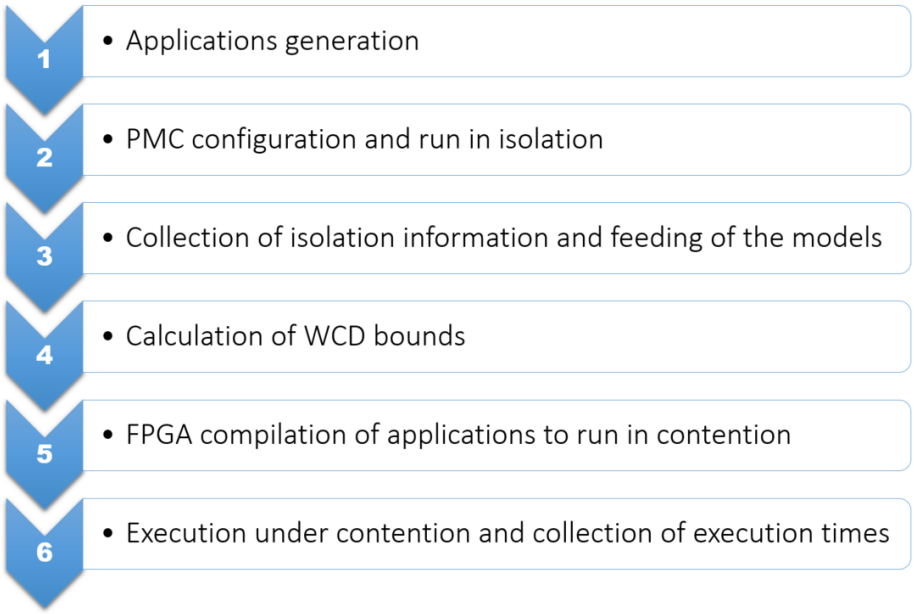


Figure 7.7: Board experiment flow

1. As a first step, an automatic code generator had been configured to generate code simulating application tasks, up to (approximately) half of a MIF utilization (i.e., 50ms). The generated code included both tasks with evenly distributed accesses along their execution, and tasks that performed accesses to memory exclusively at the beginning and end of their execution. The output was in the form of independent C functions, which were joined up in a single sequence (to mimic the static schedule), ready to be compiled and ported to the board.

2. The PMCs of the board were configured to track the events that were of our interest. This was done by means of a library internally developed in the group. Then, each

application was compiled and executed in isolation in the FPGA, tracking the number of cycles they took to execute, as well as the bus events, using the aforementioned PMCs.

3. The collected information was provided as an input to both models, in the form of a CSV.

4. By having this information, both approaches were used to calculate their WCD bounds respectively. Additionally, the fTC budgets were computed as a reference. The iterative approach script is configured to provide this information as well, if required.

5. In the subsequent step, the applications were allocated to different memory regions in the FPGA, so that each core could start executing its application concurrently, possibly causing contention.

6. Finally, the applications were run under contention. The only output collected was the number of cycles which each core needed to execute its allocated applications when run in parallel with contenders. To do so, a hardware breakpoint was set at the very end of each application, and the number of cycles obtained when these instants were reached.

As expected, the time was significantly increased. In order to cope with the slight jitter variations which are expected to occur, which in turn could lead to slightly different task overlapping scenarios, the experiments were performed 1,000 times, simulating numerous executions of the same MIF, and enabling fluctuating jitters to play a role in each run. Actually, the variation among runs was quite insignificant as one could have expected. Furthermore, it had been accurately checked that no single MIF was overrun.

In Figure 7.8, a summary of the measured execution times versus the calculated makespans is displayed, relatively normalized to the maximum observed execution time on the board. The median execution time of these runs is presented. Both approaches as well as fTC execution times (used as a reference) are normalized to the real board execution time. For example, if the execution of one MIF took 1.5 million cycles to execute in the board, a 2 in the y-axis would mean 3 million cycles. As it can be appreciated (and as one could have expected), *ILP-WCD* is the method which provides the most realistic (tighter) WCD bounds, resulting in approximately just 2 to 2.5 times the maximum observed execution time. It is worth noting that the observed execution times are not representative of the worst-case scenario, as it is not possible to control the experiments to incur the worst-case overlapping of bus requests. However, we must consider the possibility of its occurrence, and hence the enlarged computed bounds.

With regard to the iterative approach, we can observe that it is more pessimistic than *ILP-WCD*, providing an overall budget of slightly less than 3 times the observed one in the board. Interestingly, the difference between both of the hereby presented approaches is not as much as we could have initially thought. This talks in favor of the iterative

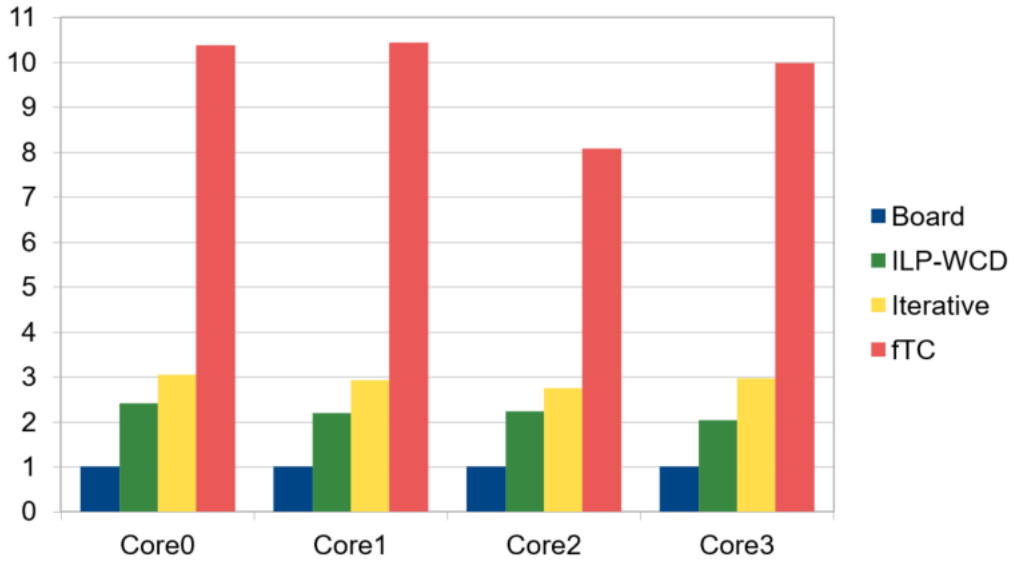


Figure 7.8: Summary of the results

method, meaning that the sources of pessimism it incurs are not excessively compromising the results, bearing in mind that the ILP solution grants us with the absolutely worst possible alignment considering the model variables.

Much more pessimistic are the fTC bounds. These results would be the ones obtained if we were only able to obtain information of the tasks run in isolation on (i) the total number of accesses that each task performs, and (ii) the longest latency experienced by these accesses.

As a final remark on the results, we want to highlight how tight the results of both approaches are. The benefits of the iterative approach with regard to task-level timing guarantees have already been observed. When it comes to the *ILP-WCD* approach, we have to bear in mind that for each core we were assuming that **every single task access** was always the last one to be granted access to the shared resource, and furthermore, it did so in the worst alignment possible, always accounting for the maximum latency. Despite the restrictive assumptions, the inflated budget (i.e., accounting for the WCD) was close to 2x the worst-case observed timing on the board, which are inherently far from the worst case. A relatively small difference between static bounds and observed values is thus an indicator of how tight are the bounds we were able to derive on the makespan.

Chapter 8

Timing Analysis for a Space Platform Without PMU: The GR712RC

8.1 Motivation

In the preceding sections, we have discussed two approaches—iterative and ILP—for deriving worst-case delay bounds in multicore real-time systems. Both methods rely on the availability of performance monitoring units (PMUs) to obtain performance monitoring counters (PMCs) for accurate timing analysis. The iterative and ILP approaches have been illustrated with the assumption that the hardware platform under consideration is equipped with a PMU, enabling straightforward access to necessary performance data.

However, in many industrial scenarios, the target hardware may not be equipped with a PMU, presenting a significant challenge for applying these advanced timing analysis methods. This is often the case in certain space applications, where hardware constraints and legacy systems may limit the availability of modern monitoring features. To demonstrate the applicability and robustness of the iterative and ILP approaches in such constrained environments, we turn our attention to a real industrial case study involving the GR712RC board—a widely used platform in the space industry.

In this section, we will present how these two methods can still be effectively applied on the GR712RC board, a platform commonly used in space applications. The GR712RC, while robust and reliable for space missions, lacks a built-in PMU, thereby posing a unique challenge for performance monitoring and timing analysis.

The rationale behind this chapter is to illustrate a practical solution to this challenge. We will showcase how a software-based solution, implemented as part of this thesis, can be employed to derive the necessary PMCs in a platform without PMU. This solution leverages a deep understanding and analysis of the processor's architecture and bus traces to infer performance data that would otherwise be directly obtained from a PMU. By meticulously analyzing the behavior of the GR712RC's processor and its bus interactions, we can accurately estimate the performance metrics required for the iterative and ILP approaches.

We show how we successfully exploited the debug support unit to obtain the necessary information on which to build a multicore timing analysis solution. The debug support unit provides a pathway to gather detailed information about the system's behavior, similar to the data offered by PMUs. We discuss the main limitations of our approach and show their impact on timing analysis accuracy. Despite these limitations, our approach successfully identifies the number of instruction and data cache misses, their type (load or store, line fills), and their target (on-chip SRAM, off-chip SRAM, SDRAM, I/Os), providing information akin to performance monitoring counters. We also provide evidence that our approach can be deployed equally to bare-metal (BM) (i.e., no operating system) and RTOS-based systems by showing that we obtain consistent results on BM and RTEMS, a reference RTOS in the space domain.

The following subsections will detail the methodology employed to extract these performance metrics, starting with an overview of the GR712RC's architecture and its relevance to space applications. We will then delve into the software solution developed to derive the PMCs, highlighting the techniques used to analyze bus traces and infer the required performance data. Finally, we will demonstrate the results of the timing analysis performed using the derived PMCs, validating its effectiveness and reliability in the absence of a physical PMU.

Through this industrial case study, we aim to provide a comprehensive guide for applying advanced timing analysis methods in environments where traditional hardware support for performance monitoring is unavailable. This not only extends the applicability of these methods but also ensures that critical systems, such as those used in space applications, can benefit from rigorous and accurate timing analysis despite hardware limitations.

It is worth noting that the study presented in this chapter was effectively and successfully used in an industrial project with a major company from the space sector.

8.2 Target Platform

The GR712RC board by Cobham Gaisler is a common choice for space missions [9] and it is the target of our space case study. The GR712RC is a radiation-hard-by-design board intended for aerospace applications. It integrates a dual-core LEON3 processor (see Figure 8.1), which is connected to the rest of the on-chip devices via a high-bandwidth AMBA AHB Bus.

Other components such as the UART are first connected to a low-bandwidth AMBA APB Bus that is a master to the AMB AHB Bus. Each core has its private instruction and data caches, with the rest of the processor resources, such as memories and peripherals, shared between both cores. For our study, we are particularly interested in capturing accesses to the on-chip SRAM, the off-chip SRAM and SDRAM, and the UART, as they are the resources used by the case study application. We target at deriving contention bounds by tracking read/write operations to these resources. Unlike other similar systems-on-chips (SoC), the GR712RC lacks a PMU, which is in charge of collecting processor usage statistics, e.g. related to the memory accesses performe and instruction types executed by

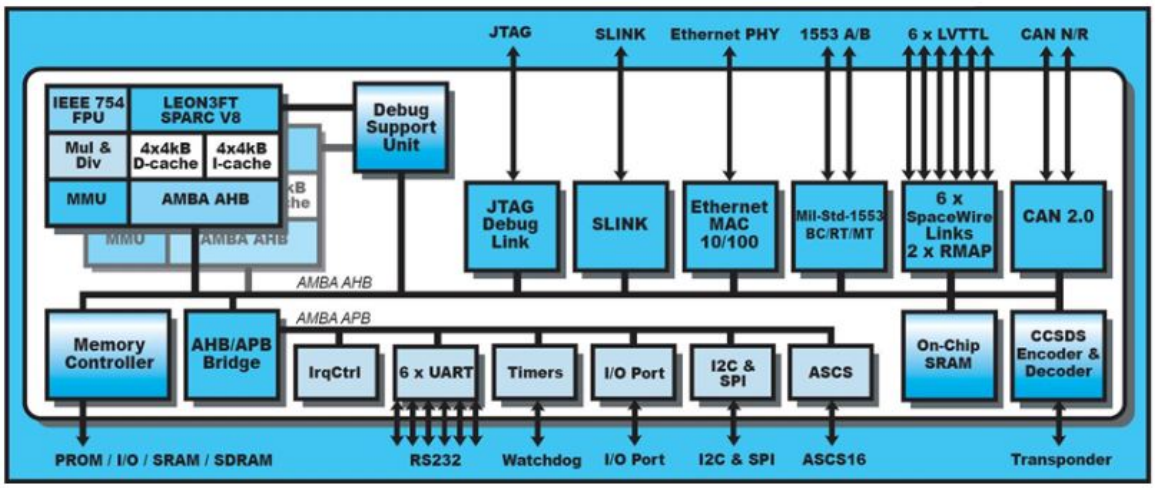


Figure 8.1: Block Diagram of the GR712RC [36]

the core. The decision of not adding a PMU is a desig choice of the hardware manufacturer and can be related to area or cost constraints. The GR712RC comes equipped with a DSU that can be accessed connecting the GR712RC board to a host using a JTag cable. We used GRMON [37], a debug tool provided by Cobham Gaisler, to connect a host computer to the DSU and issue debug commands to it, such as to extract data from the ITB and the BTB. The processor must be stopped before the traces can be extracted. We use breakpoints and step-by-step execution to stop the processor and issue debug commands, including trace extraction.

8.3 Proposed Profiling Solution

Our proposed profiling approach is designed to collect end-to-end observations using a limited set of monitoring counters. Unlike traditional methods, our solution does not require associating events directly with timing since we analytically model the impact of contention on timing using event information. This separation allows us to gather timing information and event counts independently.

The profiling solution comprises both online and offline components, as illustrated in Figure 8.2. The online component involves a GRMON script that loads an image and collects the BTB and ITB (Dumper). The offline component includes a Merger function that processes the output file from the Dumper, filtering out redundant entries and separating data into ITB and BTB structures. This step also involves merging traces to identify the instructions generating each bus request. Subsequently, the post-processing script (Collector) derives the access counts of each type.

Before detailing the solution, we highlight two key features of the GR712RC board and our contention modeling approach:

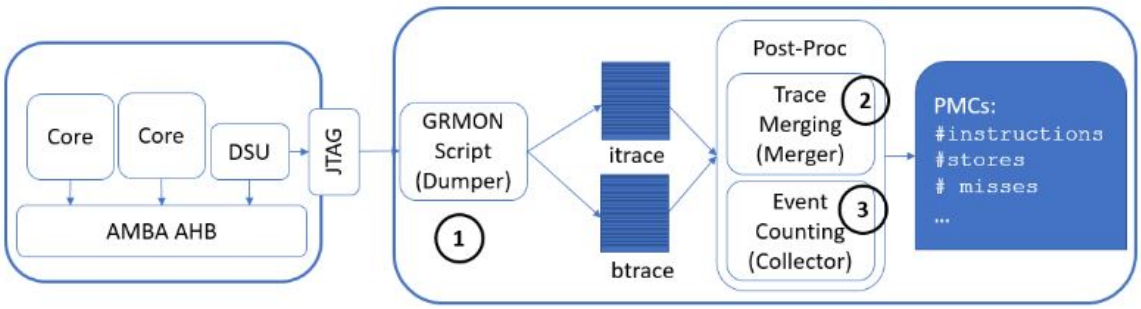


Figure 8.2: Trace Collection Process

1. On the GR712RC, the number of memory requests made by a task in isolation is the same as when it co-runs with other tasks, assuming task independence. While the latency of each request changes due to task integration, the total number of requests remains constant.

2. Our multicore timing model considers the worst-case interleaving of requests between the analyzed task and its contender.

The first feature allows us to profile access counts for each application running in isolation, eliminating the need for multicore executions. The second feature ensures that the timing of requests is irrelevant, allowing our solution to capture end-to-end access counts without affecting the timing between requests.

To address the lack of a PMU, we utilize the Debug Support Unit (DSU) features. The DSU is connected to the main AHB Bus, which links cores, debug I/O, and on-chip/off-chip memories, and can issue commands and receive information directly from the cores. It provides debugging capabilities such as breakpoints, step-by-step execution, and memory inspection. The DSU offers two key features for our study:

1. The DSU snoops the AHB bus to capture core-initiated bus activity, providing a trace of events recorded when they occur.

2. The DSU captures a stream of executed instructions in each core, recording information such as the instruction and its program counter (PC).

In the bus trace, we can identify the source of activity via the AHB MASTER field, which uniquely identifies the request's origin. Different memories are mapped to distinct address ranges, allowing us to determine the target memory for each event. Similarly, we can identify accesses to peripherals like the UART.

This data is stored in two circular buffers: the AHB Bus Trace Buffer (BTB) and the Instruction Trace Buffer (ITB). The BTB records bus activity, while the ITB logs the instruction execution stream from the cores. These buffers fill quickly, and since the DSU is connected via the shared AHB bus, the buffers must be dumped before they overflow to avoid data loss. Each buffer can hold 256 entries.

Each BTB entry includes: a timestamp, the PC of the instruction causing the bus transaction, target memory address, data, access type (read/write), and AMBA protocol options such as AHB HTRANS, AHB HBURST, AHB HSIZE, AHB HMASTER, AHB MASTLOCK, and AHB HRESP [36]. Detailed information about these fields is available in the AMBA AHB specification [12].

Each ITB entry includes: a timestamp, PC, instruction word (iword), result (or data for memory instructions), and flags indicating traps (debug mode), processor error mode, and single/multicycle instructions [36].

We demonstrate the effectiveness of this profiling approach, which successfully identifies instruction and data cache misses, their types (load/store, line fills), and targets (on-chip SRAM, off-chip SRAM, SDRAM, I/Os). This information, similar to performance monitoring counters, enables accurate timing analysis in both bare-metal (BM) and RTOS-based systems. Our method has been validated with consistent results on BM and RTEMS, a reference RTOS in the space domain.

8.3.1 Dumper

The trace buffer data fields cannot be collected in real-time, necessitating regular dumps to prevent buffer overruns. This task is managed by the **Dumper**, which operates by setting breakpoints in the region of interest and executing the program step-by-step. The Dumper is a TCL script that issues GRMON commands and runs when the GRMON debugger connects the host to the board.

Initially, the Dumper loads the executable binary onto the board, setting the entry point and stack pointer to fixed addresses in bare-metal (BM) environments. It then sets a breakpoint at the start of the region to be traced, which can encompass an entire task or a specific section. The processor boots and executes the binary normally until it hits the breakpoint. Control then shifts back to the Dumper, which resumes the program using step-by-step execution.

Every 16 steps, the DSU halts execution, and the contents of the BTB and ITB are dumped to the host via UART. The choice of 16 steps is conservative enough given the buffer sizes (256 entries each). This interval may cause some trace buffer entries to be dumped multiple times, but the Merger later filters out these duplicates. Reducing the step size further would unnecessarily slow down the process.

This dumping process continues until the program counter reaches the end of the region of interest, at which point the Dumper stops the program execution and saves the dumped file. The output is a plain text file, consisting of alternating blocks of 256 ITB entries and 256 BTB entries.

8.3.2 Merger

Once the region of interest is fully traced and the output file saved, the data undergoes **Post-processing** on the host machine.

The first step involves the **Merger**, where the btrace and itrace data are filtered

into two separate data structures using Pandas, an open-source data analysis tool built on Python. During this phase, the Merger also eliminates redundant entries caused by the conservative tracing dump rate. Next, the opcode is extracted from the iword field to determine the instruction type.

In the second step, the Merger links entries in the btrace data structure, specifically those corresponding to data load misses in the cache, to the load instructions that triggered the bus activity. This is achieved by filtering bus events whose opcode corresponds to a load instruction, the AHB HTRANS field is marked as non-sequential, and the AHB HBURST field is marked as single. These are then matched with entries in the instruction data structure that correspond to a load instruction executed one cycle after being traced on the bus. This "tracing relationship" for load miss cycle differences was empirically confirmed through multiple experiments, consistently following this pattern. Instruction misses are identified and related in a similar manner.

8.3.3 Collector

Finally, once the data structures are filtered and properly processed, the **Collector** derives the counts of events for each type of access. This process results in the events listed in 8.1.

Table 8.1: Collected Events

Event ID	Event Description
`icount`	Total number of instructions executed
`LDc`	Total number of load instructions executed
`STc`	Total number of store instructions executed
`LDm`	Total number of data cache load misses
`LDh`	Total number of data cache load hits
`Ihits`	Total number of instructions cache hits
`Imiss`	Total number of instructions cache misses
`Ifill`	Total number of instructions snooped as a result of an instruction miss
`Bus`	Total number of bus accesses
`LD_SDRAM`	Total number of loads from the SDRAM memory
`LD_offSRAM`	Total number of loads from the off-chip SRAM memory
`LD_onSRAM`	Total number of loads from the on-chip SRAM memory
`LD_IO`	Total number of loads/reads from the I/O peripherals
`ST_SDRAM`	Total number of stores to the SDRAM memory
`ST_offSRAM`	Total number of stores to the off-chip SRAM memory
`ST_onSRAM`	Total number of stores to the on-chip SRAM memory
`ST_IO`	Total number of stores/writes to the I/O peripherals

The number of executed instructions is determined by counting the entries in the instruction data structure. Instruction cache misses are identified by matching the *address* field in both traces and verifying that the access mode in the bus data structure is read

(indicating an instruction cache miss) and the value of the AHB HTRANS field is *non-sequential*. Conversely, instruction fetches resulting in a cache line fill are identified in the bus data structure by a *sequential* value for the AHB HTRANS field and an *incremental* value for AHB HBURST. These typically occur as a set of 7 instructions fetched after an instruction miss, consistent with the size of an instruction cache line. The rationale is that an executed instruction traced in the btrace implies that the instruction was not found in the instruction cache and therefore had to be fetched.

We derive the instruction cache hit count by counting the number of executed instructions that do not cause bus activity. This is achieved by subtracting the sum of instruction misses and instruction line fills from the total instruction count.

The total number of load and store instructions executed is derived from the data structure processed by the Merger, which has previously decoded the instruction type. Data cache load misses are identified as described earlier in Section 8.3.

Data cache load hits are derived by subtracting these misses from the total number of loads.

The total number of bus accesses, as well as their target memory, is derived from the bus data structure by checking the *address* field for load instructions, as each target is mapped to a different address range. The target memory of store instructions can be extracted from the instruction data structure, specifically from its *result* field, which contains the target address.

8.4 Validation of the Profiling Solution

The experimental validation of the profiling solution serves three main purposes. First, we aim to demonstrate the accuracy of the implemented profiling library (Section 8.4). Second, we utilize this library to characterize the timing impact of contention when accessing shared memory devices on the GR712RC (Section 8.5). Finally, we use this timing characterization to derive a preliminary model for a potential space application case study (Section 8.6).

To assess the accuracy of the proposed profiling solution, we use specific code snippets known as microbenchmarks (uBs), as discussed in Section 4. This validation covers two primary dimensions:

1. We compare the expected access count values with those obtained using our profiling solution in a bare-metal setup. This provides an assessment of the solution's accuracy in a pristine environment.

2. We compare the results obtained from the profiling solution when running the same code snippets under both bare-metal and RTEMS environments. This helps identify any portability issues of the solution across different RTOS on the GR712RC.

For this validation, the code snippets are designed so that a hardware expert, with an understanding of the GR712RC architecture, can provide highly accurate estimates of the expected access counts. These snippets are also small enough to be reasonably manageable

by the expert. Previous explorations on using specialized code snippets for characterizing multicore timing interference have been reported in [66]. The basic structure of each snippet consists of a main loop containing a large body of one or two types of instructions, usually load and/or store. This design minimizes the overhead from loop control instructions and, by varying the range of addresses accessed by the load/store operations, forces accesses to target on-chip SRAM, off-chip SRAM/SDRAM, or the UART

Table 8.2: List of code snippets

Name	Description
cs_ic_hit	Causes instruction cache hits
cs_ic_miss	Causes inst. cache misses and line refills
cs_dc_hit	Causes data cache load hits
cs_on_SRAM_rd	Causes on-chip SRAM read accesses
cs_on_SRAM_wr	Causes on-chip SRAM write accesses
cs_off_SRAM_rd	Causes off-chip SRAM read accesses
cs_off_SRAM_wr	Causes off-chip SRAM write accesses
cs_off_SDRAM_rd	Causes off-chip SDRAM read accesses
cs_off_SDRAM_wr	Causes off-chip SDRAM write accesses
cs_UART_rd	Causes UART read accesses
cs_UART_wr	Causes UART write accesses

The range of accessed addresses can also be adjusted to ensure these accesses cause capacity misses in the cache. The high density of desired operations in these snippets, combined with visual inspection of the object code, allows the hardware expert to predict the expected behavior concerning the most relevant events with high accuracy.

We validate the accuracy of the proposed solution by applying it to the code snippets listed in Table 8.2. These experiments are conducted in a single-core scenario as specified in Section 4. We compare the collected values to those expected. Due to space limitations, we present the validation for a selection of snippets, excluding *cs_ic_miss* and *cs_ic_hit*. Each validation experiment highlights only the most relevant events.

The experimental setup involves running the code snippets in a controlled environment on the GR712RC board. The profiling solution's accuracy is assessed by comparing the observed and expected values for key performance metrics. The snippets are carefully crafted to generate predictable behavior, allowing for precise validation of the profiling approach. The validation focuses on critical events, such as instruction cache misses and hits, to ensure the profiling solution accurately captures these events in both bare-metal and RTEMS environments.

8.4.1 Bare Metal Microbenchmarks

cs_dc_hit. This microbenchmark is designed to generate loads that consistently hit the L1 data cache. Before profiling begins, the loop function is executed to warm up the caches and

minimize cold misses, which target the off-chip SRAM memory. Each iteration of the loop, which runs 1000 times, executes a total of 128 load instructions. Table 8.3 demonstrates the high accuracy of our profiling library in reporting instruction count (ICount), load and store counts (LDc and STc), load hits and misses (LDh and LDm), instruction cache hits (Ihits), and accesses to both the off-chip SDRAM (SDRAM) and the on-chip SRAM (oSRAM). The maximum deviation observed is 0.03% for instruction counts. Such minimal deviations are commonly observed in many COTS platforms, even in single-core, non-speculative ones. Another potential cause could be the rotating trace buffer, which might include requests slightly before and after the region of interest in the trace dump. The 23 accesses to the off-chip SRAM reported in Table 8.3 are attributed to cache line refills, which are also captured by the post-processing script. For the instruction hits (Ihits) count, we expect all instructions (Icount) to hit in the L1 instruction cache. Further experiments indicated that increasing the loop iteration count reduces the relative deviation, suggesting a constant instruction count overhead. Additionally, the obtained counts are deterministic across multiple runs.

cs_X_rd. Table 8.4 shows the derived and expected access counts for snippets designed to systematically miss load operations from the L1 cache, targeting different memory types: cs_on_SRAM_rd, cs_off_SRAM_rd, and cs_off_SDRAM_rd.

Table 8.3: Validation of the profile solution with `cs_dc_hit`.

Event	**Exp.**	**Obs.**	**Dev (%)**
Icount	131000	131040	0.03
LDc	128000	128004	0.00
STc	0	1	-
LDm	0	1	-
LDh	128000	128003	0.00
Ihits	131040	131036	0.00
LD SDRAM	0	0	0.00
LD offSRAM	0	23	-
LD onSRAM	0	0	0.00
ST offSRAM	0	1	-

To eliminate any residual data in the cache from previous executions, the loop functions are executed before profiling begins. Similar to cs_dc_hit, the loop function comprises 1000 iterations, each performing 128 load operations to the specified memory, with a stride between them ensuring each load instruction causes a miss in the L1 data cache. Results for all three memories confirm very high accuracy for the relevant events, with the worst case deviation (0.06%) again associated with instruction counts. The 43 loads from the off-chip SRAM in cs_loadmiss_onsram include instruction misses and instruction cache line refills triggered by an instruction cache miss.

cs_X_wr. We also validated the uB that perform write operations to different memory devices on the board (cs_on_SRAM_wr, cs_off_SRAM_wr, and cs_off_SDRAM_wr). These

Table 8.4: Expected & Observed event counts, and relative Deviation (%) for `cs_X_rd`.

	OnSRAM			**OffSRAM**			**SDRAM**		
Event	**Exp.**	**Obs.**	**Dev (%)**	**Exp.**	**Obs.**	**Dev (%)**	**Exp.**	**Obs.**	**Dev (%)**
Icount	131000	131073	0.06	131000	131024	0.02	131000	131024	0.02
LDc	128000	128000	0.00	128000	128001	0.00	128000	128001	0.00
STc	0	1	-	0	0	-	0	0	-
LDm	128000	128003	0.00	128000	128000	0.00	128000	128002	0.00
LDh	0	3	-	0	0	-	0	0	-
Ihits	131073	131061	0.01	131073	131064	0.01	131073	131064	0.01
LD SDRAM	0	0	0.00	0	0	0.00	0	0	0.00
LD offSRAM	0	43	-	0	43	-	0	43	-
LD onSRAM	128000	128000	0.00	0	0	-	0	0	-
ST offSRAM	0	1	-	0	1	-	0	1	-

snippets consist of 1000 loop iterations, each triggering 128 store operations to addresses in the respective memory. Due to the write-through no-allocate policy, caches do not need pre-heating, as each store results in a bus access without loading content into the L1 cache. Accuracy results, which are not shown due to space constraints, are consistent with those observed for read operations.

cs_UART_X. For the UART uB, which involve reading from or writing to an I/O device, we first configure the UART registers and then perform reads or writes in a loop composed of 128 accesses to the memory-mapped address in the UART registers. Results confirm the accuracy of our tool in tracing and collecting events, with a maximum deviation of 0.11% in the instruction count.

8.4.2 RTEMS Microbenchmarks

One of the requirements for our profiling solution is its ability to support various real-time operating systems with minimal modifications. This flexibility is crucial since some case studies run on bare metal (BM) while others use established RTOS platforms. To demonstrate this capability, we evaluated our profiling method with the Real-Time Executive for Multiprocessor Systems (RTEMS) v5.

Changes in the Profiler. Adapting our solution to RTEMS did not necessitate any changes to the profiling scripts. The steps for tracing and post-processing remain the same as those used for bare metal. At the RTEMS level, the only required configuration is either to use a uniprocessor scheduler or to disable the other potentially contending cores when profiling an application using a multicore scheduler. Both configurations allow tracing in isolation, as required by our profiler.

Changes in the Code Snippets. When developing the snippets for RTEMS, our objective was to ensure that the same binary could run on both BM and RTEMS. This was achieved by compiling the snippet into a .o object file using the sparc-elf-gcc cross-compiler and then linking it into either a BM image or an RTEMS image using the sparc-elf

or sparc-rtems tool-chain linker. To enable the use of the same object file for both BM and RTEMS, the code snippet was built without any library or system call dependencies.

Evaluation. The evaluation process for RTEMS was analogous to that for bare metal, involving a comparison between expected and observed counts. The region of interest is the loop function of the code snippets, i.e., after RTEMS has already been initialized. Therefore, we do not expect significant variance in the results due to interference by the RTOS. Table 8.5 reports the expected and observed results for the snippets reading from different memories under RTEMS. The results show very similar values, with a slight difference in the number of instructions executed by the processor (0.1%) and the load count (0.02%). These differences, which can be attributed to the RTOS, are considered negligible. We also conducted the same experiments for other code snippets, whose results are not shown due to space constraints, and observed similar conclusions: a maximum of 0.12% deviation in Icount and no deviation in STc for all the *cs_*_wr* snippets. The results demonstrate a very small deviation between expected and observed values, even in the presence of an RTOS.

Additionally, since the same code runs on both BM and RTEMS and we trace the same region of interest, we can directly compare the results under BM and RTEMS, as shown in Table 8.4 and Table 8.5. The fact that the tool-chain can be applied to RTEMS without any modifications to the collector code or the dumper and merger scripts indicates that this process can be easily extended to support other RTOS platforms. This flexibility is crucial for ensuring that our profiling solution is robust and adaptable to various industrial scenarios.

Table 8.5: Validation with memory read snippets in RTEMS.

	OnSRAM			**OffSRAM**			**SDRAM**		
Event	**Exp.**	**Obs.**	**Dev (%)**	**Exp.**	**Obs.**	**Dev (%)**	**Exp.**	**Obs.**	**Dev (%)**
Icount	131000	131136	0.10	131000	131136	0.10	131000	131136	0.10
LDc	128000	128022	0.02	128000	128022	0.02	128000	128022	0.02
STc	0	0	-	0	0	-	0	0	-
LDm	128000	128000	0.00	128002	128002	0.00	128000	128002	0.00
LDh	0	20	-	0	20	-	0	1	-
Ihits	131136	131129	-0.01	131136	131129	-0.01	131136	131129	-0.01
LD SDRAM	0	0	0.00	0	0	0.00	0	0	0.00
LD offSRAM	0	44	-	128000	128045	0.04	0	44	-
LD onSRAM	128000	128000	0.00	0	0	0.00	0	0	0.00

8.5 Results Evaluation: Contenion Slowdown Matrix

At this stage, we have already discussed that contention models typically rely on access counts, which can be obtained using our profiling tool as demonstrated in Section 8.4. Additionally, we need worst-case contention latencies for each target shared resource. For every combination of request type and target resource, we must determine the contention

impact they have on each other. This information is compiled in what we call the slowdown matrix. Each cell within the slowdown matrix is derived from running specific stressing benchmarks [25] that exert maximum load on the resource. The reliability of the slowdown matrix is thus dependent on the effectiveness of these stressing benchmarks. To validate this, we must provide evidence that the benchmarks exert sufficient stress on their target resources.

Table 8.6: Stressing benchmark validation.

		Expected Relation	**Acc.**	**Validated**
On-c. SRAM	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_onSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_offSRAM = LD_offSDRAM = LD_IO = 0	100%	yes
	WR	STc ≈ Icount;	95%	yes
		STc = ST_onSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_offSRAM = ST_offSDRAM = ST_IO = 0	100%	yes
Off-c. SRAM	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_offSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_onSRAM = LD_offSDRAM = LD_IO = 0	100%	yes
	WR	STc ≈ Icount;	95%	yes
		STc = ST_offSRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_onSRAM = ST_offSDRAM = ST_IO = 0	100%	yes
Off-c. SDRAM	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_offSDRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_offSRAM = LD_onSRAM = LD_IO = 0	100%	yes
	WR	STc ≈ Icount;	95%	yes
		STc = ST_offSDRAM = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_offSRAM = ST_onSRAM = ST_IO = 0	100%	yes
UART	RD	LDc ≈ Icount;	97%	yes
		LDc = LDm = LD_IO = Bus	100%	yes
		Ihits = Icount	100%	yes
		LD_offSRAM = LD_onSRAM = 0	100%	yes
	WR	STc ≈ Icount;	97%	yes
		STc = ST_IO = Bus	100%	yes
		Ihits = Icount	100%	yes
		ST_offSRAM = ST_onSRAM = 0	100%	yes

In our experimental setup, we utilize our profiling solution to effectively meet this requirement. While the objective for code snippets was to verify that expected absolute event counts matched the observed counts, for stressing benchmarks, the focus is on ensuring a specific relationship between event counts. For example, a read-stressing benchmark targeting the on-chip SRAM should exhibit: (1) a match between the number of instructions and load operations; (2) a correlation between data cache hits, data cache misses, and on-chip SRAM accesses with load operations. The first condition captures that, apart from a few control instructions in the main loop, the majority of instructions should correspond to the target type. A few additional instructions might be added to prevent systematic behaviors [35]. The second condition indicates that read operations should miss in the data cache and access the on-chip SRAM.

				Contender							
				On-chip SRAM		Off-chip SRAM		SDRAM		UART	
			Isol.	RD	WR	RD	WR	RD	WR	RD	WR
Analysis	On-c. SRAM	RD	7	9.0	8.5	11.0	10.0	12.0	8.1	9.0	8.0
		WR	2	4.3	3.0	4.5	7.0	5.0	6.1	3.5	5.0
	Off-c. SRAM	RD	8	11.0	9.0	12.0	11.0	13.0	11.1	10.0	9.0
		WR	6	10.0	7.0	11.0	11.0	13.0	11.8	9.0	9.0
	SDRAM	RD	9	12.0	10.1	13.0	13.0	14.1	13.2	11.1	10.1
		WR	6	8.0	6.1	11.0	12.0	13.1	12.1	7.1	7.1
	UART	RD	6	9.0	7.0	10.0	9.0	11.1	7.1	8.0	7.0
		WR	4	8.0	5.0	9.0	9.0	10.0	7.1	7.0	7.0

Figure 8.3: Derived Slowdown Matrix for the GR712RC [number of cycles]

Figure 8.3 presents the uBs developed, along with the expected relationships among event counters, and the evaluation results based on observed event counts using our profiling tool. The results confirm that the stressing benchmarks effectively stress their target resources, providing credibility to the results obtained in the Slowdown Matrix. The first column of Figure 8.3 reports the number of clock cycles required to execute each type of instruction in isolation. The subsequent columns indicate the number of cycles needed for the same instruction when another core is simultaneously executing a particular instruction intensively.

8.6 Case Study: Space Application

To assess the efficacy of our profiling approach, we applied it to a real-world space application that implements a subset of telemetry and telecommand (TM/TC) functionalities. The TM/TC subsystem is a crucial component of a spacecraft that serves two primary purposes: **(i)** collecting and transmitting measurements (telemetry) to ground-based facilities, and **(ii)** receiving commands (telecommand) from ground control to man-

age and operate the spacecraft during its development, assembly, integration, testing, launch phases, and subsequent operation. Typically, the TM/TC subsystem interfaces with various other spacecraft subsystems through diverse communication channels such as CAN buses, SpaceWire, SpaceFiber, or RS-232 connections, while the ground communication relies on RF baseband interfaces.

For the scope of this study, we focused on three specific applications within the TM/TC subsystem, particularly those handling UART communication for I/O operations.

The first application is a **Watchdog** system, designed to ensure continuous service for an unmanned system, which cannot be manually operated in case of failure. The watchdog application monitors the health of other co-running applications within the TM/TC subsystem, rebooting the hardware board if any application's service is disrupted due to issues such as infinite loops, software crashes, or unhandled exceptions. The watchdog leverages hardware timers available on the GR712RC board, setting a counter that decrements progressively and triggers a board reboot if it reaches zero. Meanwhile, the other applications periodically send keep-alive signals to the watchdog to reset the timer, ensuring all systems are operational.

The second application, the **Scrubber**, is tasked with maintaining the reliability of the memory subsystem in a radiation-prone environment. Its function is to exercise and refresh the ECC (Error-Correcting Code) protection of the GR712RC's off-chip SRAM memory. The scrubber sequentially reads the entire 8MB SRAM memory in 64B blocks and, upon detecting an ECC correction, rewrites the word back to memory to prevent further bit shifts from rendering the error uncorrectable. This application performs non-cacheable, word-by-word load accesses to the memory and verifies each load against the AHB status register to detect any ECC corrections.

Finally, the **Crypter** application handles the AES (Advanced Encryption Standard) encryption and decryption of telemetry data and ground commands. The crypter utilizes a symmetric block cipher based on a substitution-permutation network, which involves several rounds of computation to ensure data security.

Through this evaluation, we aim to demonstrate the applicability and robustness of our profiling solution in a real industrial setting, particularly for critical space applications where reliability and accuracy are paramount.

8.7 Contention Results

Following the methodology outlined in the previous section, we derived the event counts for three applications, resulting in the profiles presented in Table 8.7. These profiles formed the basis for our fully Time-Composable (fTC) execution time estimates.

The fTC estimates assume that each request generated by the application contends with a concurrent request, such that both requests arrive simultaneously, but the application's request always loses the arbitration, thus suffering the maximum possible contention. For example, if the application generates a read to the on-chip SRAM, a contender generates a read to the off-chip SDRAM in the same cycle and wins the arbitration, causing

the application's request to experience a contention delay of 12.0 cycles.

Figure 8.4 illustrates a comparison between the fTC estimates and the observed execution time when each application is run against a stressing contender, referred to as the multicore scenario execution time (ETmuc). As anticipated, the fTC estimates are consistently higher than the ETmuc. For the Crypter and Scrubber applications, we observed a modest overestimation ranging from 1.25 to 1.35 for Crypter and from 1.01 to 1.11 for Scrubber. This overestimation is attributed to the assumed alignment between the application's request and the contender's request. In practice, it is challenging, if not impossible, to perfectly align the requests of an arbitrary application with those of its contender. However, the fTC model assumes this worst-case scenario, leading to some level of overestimation, albeit limited.

For the Watchdog application, which has a relatively short execution time of 201 cycles in isolation, the impact of the worst-case alignment assumption is more pronounced in relative terms. However, this overestimation is insignificant when considering the overall execution time. Figure 8.4 displays the absolute execution time of each application when running in isolation (ETisol), represented by the bars on the left vertical axis. The line represents the ratio of fTC estimates to ETisol, showing the relationship between the estimated execution time and the execution time in isolation. For Crypter and Scrubber, this ratio remains below 1.5x. However, for the Watchdog, due to its brief execution time, the ratio exceeds 3.0. This is because any small overestimation of the fTC estimate becomes significant in relative terms, despite being minor in absolute terms.

Table 8.7: Resulting profiles for the considered TM/TC sub-system elements.

Event	**Crypter**	**Scrubber**	**Watchdog**
Icount	2057	539	26
LDc	513	132	5
STc	121	2	0
LDm	312	132	5
LDh	0	0	0
Ihits	1882	531	19
Imiss	176	8	7
I line fill	39	12	6
LD SDRAM	0	0	0
LD offSRAM	531	87	27
LD onSRAM	0	0	0
LD IO	0	0	0
ST SDRAM	0	0	0
ST offSRAM	121	65	0
ST onSRAM	0	0	0
ST IO	0	0	1

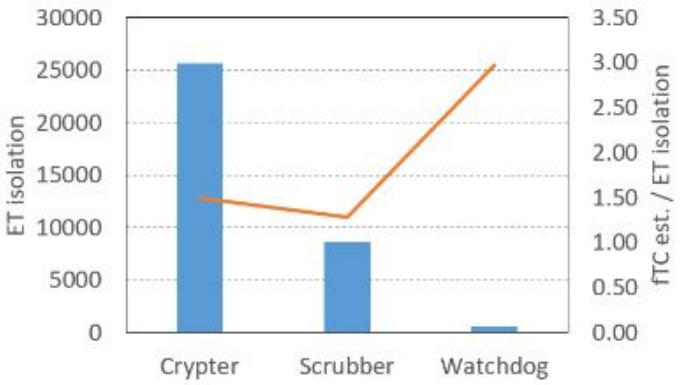


Figure 8.4: Comparison to ETisol

Chapter 9

Conclusions

This thesis makes significant contributions to the field of real-time multicore timing analysis, addressing key challenges in determining Worst-Case Execution Time (WCET) in multicore systems. By proposing novel methodologies, practical tools, and real-world applications, this work bridges the gap between theoretical research and industrial requirements. This chapter consolidates the key contributions, highlights the significance of the results, and outlines potential avenues for future research.

Contributions and Results

The core contributions of this thesis span theoretical innovations, practical methodologies, and their validation in industrial scenarios. The primary achievements are summarized as follows:

1. **Iterative Method for Task-Level WCD Bounds:** This method provides task-level guarantees by iteratively refining WCD estimates for individual tasks. By modeling interactions with potential contenders and reducing unnecessary pessimism, the approach achieves tight and safe bounds while maintaining scalability.

2. **ILP-Based System-Level Approach:** An Integer Linear Programming (ILP)-based methodology was developed to compute system-level WCD bounds. By capturing task interactions across the entire scheduling makespan, this approach produces tighter and more representative bounds, especially in complex multicore scenarios.

3. **Automated Framework:** A fully automated framework was implemented to support the proposed methodologies. This framework generates executable Python code tailored to multicore configurations, significantly simplifying deployment and reducing manual errors.

4. **Software Solution for Non-PMU Systems:** For hardware platforms lacking Performance Monitoring Units (PMUs), a software-based solution was developed to

derive critical performance metrics. By analyzing bus and instruction traces, this approach enables accurate profiling and timing analysis even in constrained systems.

5. **Design and Implementation of Microbenchmarks:** Custom microbenchmarks were created to validate the profiling framework, stress-test hardware components, and generate high-density events for analysis. These benchmarks play a dual role in verifying profiling accuracy and serving as controlled workloads for timing analysis.

6. **Industrial Case Study:** The proposed methods were applied to a real-world industrial project in the space domain, using the GR712RC board. This case study validated the safety, precision, and industrial applicability of the proposed approaches.

7. **Comparative Evaluation:** Rigorous evaluations demonstrated the strengths of the proposed methods. The iterative approach achieved overestimation margins below 1.5x for most tasks, while the ILP-based approach produced even tighter bounds for system-level scenarios.

These contributions collectively advance the state of the art in multicore timing analysis, offering robust, scalable, and practical solutions for real-world applications.

Future Directions

This work opens several avenues for future research and development, building upon the methodologies and insights presented:

1. **Refined Task Overlap Modeling:** Future work could quantify the extent and duration of task overlaps, moving beyond binary overlap assumptions. This refinement could yield more realistic contention models and tighter bounds.

2. **Event Distribution and Ordering:** Incorporating knowledge about the sequence and distribution of task requests could further refine contention models. Analyzing tasks in phases or segments could eliminate unrealistic collision scenarios and improve bound tightness.

3. **Enhanced Profiling Capabilities:** Expanding profiling frameworks to include additional performance counters or advanced profiling techniques could improve access classification, reducing conservatism in WCD estimates.

4. **Dynamic Scheduling Scenarios:** Extending the proposed methodologies to dynamic scheduling paradigms and other multicore architectures would broaden their applicability and relevance to modern systems.

By addressing these challenges, future research can further advance the field of multicore timing analysis, ensuring safe and efficient deployment of multicore systems in critical applications.

Bibliography

[1] Cyclic executive scheduling. https://pubweb.eng.utah.edu/~cs5785/slides-f10/22-1up.pdf.

[2] IBM CPLEX solver. https://www.ibm.com/analytics/cplex-optimizer.

[3] Leon4 datasheet. https://www.gaisler.com/doc/LEON4-N2X-DS.pdf.

[4] Leon4 summary. http://www.sjalander.com/research/pdf/sjalander-dasia2009.pdf.

[5] SYSGO PikeOS RTOS. http://www.sysgo.com/, 2018.

[6] WIND RIVER VxWorks MILS Platform 3.0. https://www.windriver.com/, 2018.

[7] A. Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296, April 2015.

[8] Abderaouf N. Amalou, Isabelle Puaut, and Gilles Muller. WE-HML: Hybrid WCET Estimation Using Machine Learning for Architectures with Caches. In *IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 31–40, 2021.

[9] Tadeu Nogueira C. Andrade, George Lima, Veronica Maria Cadena Lima, Slima Bem-Amor, Ismail Hawila, and Liliana Cucu-Grosjean. On the Impact of Hardware-Related Events on the Execution of Real-Time Programs. In *Design Automation for Embedded Systems*, volume 27, page 275–302, 2023.

[10] ARINC. *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc, 1997.

[11] ARM. ARM® CoreSight© IP. https://www.arm.com/products/system-ip/coresight-debug-trace.

[12] ARM. *ARM Advanced Microcontroller Bus Architecture (AMBA) 5 AMBA High-performance Bus (AHB) Protocol Specification*. ARM Ltd., 2015. Available online: https://developer.arm.com/documentation (accessed 2015).

[13] AUTOSAR. *Specification of RTE Software - AUTOSAR CP Release 4.3.1*, 2017.

[14] A. Baldovin, E. Mezzetti, and T. Vardanega. A time-composable operating system. In *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012*, pages 69–80.

[15] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nelis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, July 2016.

[16] Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. Multi-core composability in the face of memory-bus contention. *ACM SIGBED Review*, 10(3):35–42, 2013.

[17] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005.

[18] A. Biondi and M. Di Natale. Achieving predictable multicore execution of automotive applications using the LET paradigm. In *24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018.

[19] Shirley Browne, Jack J. Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[20] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*.

[21] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1068–1075, Nov 2011.

[22] D. Dasari and V. Nelis. An Analysis of the Impact of Bus Contention on the WCET in Multicores. In *IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems*, pages 1450–1457, 2012.

[23] Dakshina Dasari, Bjorn Andersson, Vincent Nelis, Stefan M Petters, Arvind Easwaran, and Jinkyu Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1068–1075. IEEE.

[24] Dakshina Dasari and Vincent Nelis. An analysis of the impact of bus contention on the wcet in multicores. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), IEEE 14th International Conference on*, pages 1450–1457. IEEE.

[25] E. Diaz, E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla. Modelling Multicore Contention on the AURIX^{TM} TC27x. In *Design& Automation Conference (DAC)*, 2018.

[26] Enrique Díaz, Mikel Fernández, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. MC2: Multicore and Cache Analysis via Deterministic and Probabilistic Jitter Bounding. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 102–118, 2017.

[27] N. Diniz and J. Rufino. ARINC 653 in Space. In *DASIA - Data Systems in Aerospace*, ESA Special Publication, 2005.

[28] Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous non-intrusive hybrid WCET estimation using waypoint graphs. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASICS*, pages 4:1–4:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[29] Boris Dreyer, Christian Hochberger, Simon Wegener, and Alexander Weiss. Precise continuous non-intrusive measurement-based execution time estimation. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 45–54, Lund, Sweden, July 7 2015. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[30] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. Scaling up the memory interference analysis for hard real-time many-core systems. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 330–333, 2020.

[31] Enrique Díaz, Jaume Abella, Enrico Mezzetti, Irune Agirre, Mikel Azkarate-Askasua, Tullio Vardanega, and Francisco J. Cazorla. Mitigating Software-Instrumentation Cache Effects in Measurement-Based Timing Analysis. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 1:1–1:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[32] Federal Aviation Administration, Certification Authorities Software Team (CAST). CAST-32A Multi-core Processors. Technical report, Federal Aviation Administration, 2016. Available upon request or through official FAA channels.

[33] G. Fernandez, J. Jalle, J. Abella, E. Quiñones, T. Vardanega, and F. J. Cazorla. Resource usage templates and signatures for cots multicore processors. In *52Nd Annual Design Automation Conference*, DAC '15, pages 155:1–155:6. ACM, 2015.

[34] Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. Introduction to partial time composability for COTS multicores. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1955–1956, 2015.

[35] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Computing safe contention bounds for multicore resources with round-robin and fifo arbitration. *IEEE Transactions on Computers*, 66(4):586–600, 2017.

[36] Cobham Gaisler. *GR712RC User Manual*, 2020. `https://www.gaisler.com/doc/gr712rc-usermanual.pdf`.

[37] Cobham Gaisler. *GRMON2 User's Manual*, 2020. `https://www.gaisler.com/doc/grmon2.pdf`.

[38] Jeremy Giesen, Pedro Benedicte, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Modeling contention interference in crossbar-based systems via sequence-aware pairing (SeAP). In *26th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020*, Sydney, Australia, April 21–24 2020. IEEE.

[39] Jeremy Giesen, Enrico Mezzetti, Jaume Abella, Enrique Fernández, and Francisco J. Cazorla. ePAPI: Performance Application Programming Interface for Embedded Platforms. In Sebastian Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[40] Sylvain Girbal, Xavier Jean, Jimmy Le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic platform software for hard real-time systems using multi-core cots. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 8D4–1–8D4–15, 2015.

[41] Sylvain Girbal, Daniel Gracia Pérez, Jimmy Le Rhun, Madeleine Faugere, Claire Pagetti, and Guy Durrieu. A complete toolchain for an interference-free deployment of avionic applications on multi-core systems. in 2015 ieee/aiaa 34th digital avionics systems conference (dasc). Technical report, pages 7A2–1–7A2–14, 2015.

[42] Sandro Grebant, Clément Ballabriga, Julien Forget, and Giuseppe Lipari. WCET Analysis with Procedure Arguments as Parameters. In *In Proceedings of the 31st International Conference on Real-Time Networks and Systems*, pages 11–22, 2023.

[43] Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjödin. The multi-resource server for predictable execution on multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 1–12.

[44] Rafia Inam, Mikael Sjödin, and Marcus Jägemar. Bandwidth measurement using performance counters for predictable multicore software. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation, ETFA 2012*, pages 1–4, Krakow, Poland, 2012.

[45] Intel. Next-Generation Transportation. `http://www.intel.com/content/www/us/en/automotive/automotive-overview.html.`, Intel Press Release, 2017.

[46] J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *8th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2016.

[47] J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla. Contention-aware performance monitoring counter support for real-time mpsocs. In *11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016.

[48] J. Jalle, M. Fernández, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F.J. Cazorla. Contention-aware performance monitoring counte support for real-time mpsocs. In *11th IEEE Symposium on Industrial Embedded Systems, SIES 2016*.

[49] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 3–12. IEEE.

[50] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, April.

[51] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 1st edition, 1997.

[52] Vikash Kumar. Estimation of an Early WCET Using Different Machine Learning Approaches. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 297–307, 2022.

[53] Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In Sebastian

Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *OpenAccess Series in Informatics (OASIcs)*, pages 1:1–1:11, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[54] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[55] Linux. perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[56] D. Locke, David R. Vogel, L. Lucas, and J. Goodenough. Generic avionics software specification. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[57] S. Martinez, D. Hardy, and I. Puaut. Quantifying WCET reduction of parallel applications by introducing slack time to limit resource contention. In *International Conference on Real-Time Networks and Systems (RTNS)*, International Conference on Real-Time Networks and Systems, October 2017.

[58] Nick Merriam, Peter Gliwa, and Ian Broster. Measurement and tracing methods for timing analysis. *International Journal on Software Tools for Technology Transfer*, 15(1):9–28, 2013.

[59] Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. High-integrity performance monitoring units in automotive chips for reliable timing V&V. *IEEE Micro*, 38(1):56–65, 2018.

[60] Nexus 5001 Forum. Nexus 5001 forum. `http://www.nexus5001.org`.

[61] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *26th Euromicro Conference on Real-Time Systems*, pages 109–118, July 2014.

[62] P. J. Parkinson. Multicore mils. In *9th IET International Conference on System Safety and Cyber Security*, pages 1–8, 2014.

[63] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

[64] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *Real-Time Systems Symposium*, pages 221–231, Nov 2008.

[65] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[66] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25.

[67] Rapita Systems. Rapita Trace Box (RTBx) Data Logger. `https://www.rapitasystems.com/products/rtbx`, 2020.

[68] Rapita Systems. Rapita Verification Suite (RVS). `https://www.rapitasystems.com/`, 2020.

[69] Federico Reghenzani, Luca Santinelli, and William Fornaciari. Dealing with Uncertainty in pWCET Estimations. In *ACM Transactions on Embedded Computing Systems*, volume 19, 2020.

[70] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In *24th International Conference on Real-Time Networks and Systems*, RTNS, pages 67–76, 2016.

[71] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[72] B. Rouxel, S. Derrien, and I. Puaut. Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1 – 20, October 2017.

[73] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Conference on Design, Automation and Test in Europe*, DATE, pages 759–764, 2010.

[74] A. Schranzhofer, R. Pellizzoni, J. J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Design Automation Conference*, pages 332–337, June 2010.

[75] Jamile Vasconcelos, George Lima, Marwan Wehaiba El Khazen, Adriana Gogonel, and Liliana Cucu-Grosjean. On Vulnerabilities in EVT-Based Timing Analysis: An Experimental Investigation on a Multi-Core Architecture. In *Design Automation for Embedded Systems*, 2023.

[76] Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.